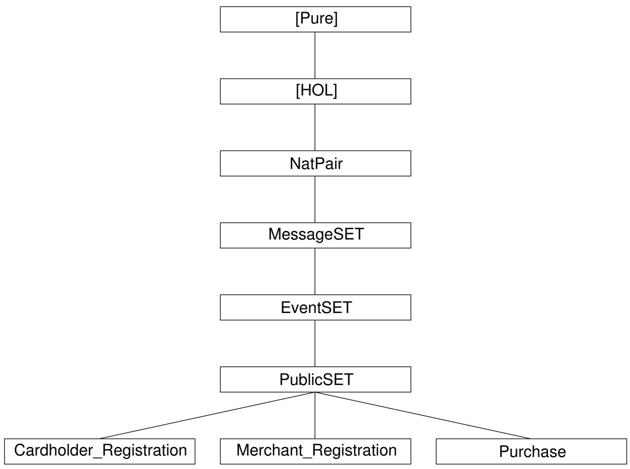
Verification of The SET Protocol

Giampaolo Bella, Fabio Massacci, Lawrence C. Paulson et al.

October 1, 2003

Contents


[Pure]
[HOL]
NatPair
MessageSET
EventSET
PublicSET
Cardholder_Registration
Merchant_Registration
Purchase

1 The Message Theory, Modified for SET

theory *MessageSET = NatPair:*

1.1 General Lemmas

Needed occasionally with *spy_analz_tac*, e.g. in *analz_insert_Key_newK*

```
lemma Un_absorb3 [simp] : "A ∪ (B ∪ A) = B ∪ A"
by blast
```

Collapses redundant cases in the huge protocol proofs

```
lemmas disj_simps = disj_comms disj_left_absorb disj_assoc
```

Effective with assumptions like *K ∉ range pubK* and *K ∉ invKey ‘ range pubK*

```
lemma notin_image_iff: "(y ∉ f‘I) = (∀ i∈I. f i ≠ y)"
by blast
```

Effective with the assumption *KK ⊆ - range (invKey ∘ pubK)*

```
lemma disjoint_image_iff: "(A <= - (f‘I)) = (∀ i∈I. f i ∉ A)"
by blast
```

```
types
  key = nat

consts
  all_symmetric :: bool          — true if all keys are symmetric
  invKey        :: "key=>key"    — inverse of a symmetric key

specification (invKey)
  invKey [simp]: "invKey (invKey K) = K"
  invKey_symmetric: "all_symmetric --> invKey = id"
    by (rule exI [of _ id], auto)
```

The inverse of a symmetric key is itself; that of a public key is the private key and vice versa

```
constdefs
  symKeys :: "key set"
  "symKeys == {K. invKey K = K}"
```

Agents. We allow any number of certification authorities, cardholders merchants, and payment gateways.

```
datatype
  agent = CA nat | Cardholder nat | Merchant nat | PG nat | Spy
```

Messages

```
datatype
     msg = Agent  agent       — Agent names
         | Number nat         — Ordinary integers, timestamps, ...
         | Nonce  nat         — Unguessable nonces
```

```
| Pan    nat        — Unguessable Primary Account Numbers (??)
| Key    key        — Crypto keys
| Hash   msg        — Hashing
| MPair  msg msg    — Compound messages
| Crypt  key msg    — Encryption, public- or shared-key
```

syntax
```
  "@MTuple"      :: "['a, args] => 'a * 'b"       ("(2{|_,/ _|})")
```

syntax *(xsymbols)*
```
  "@MTuple"      :: "['a, args] => 'a * 'b"       ("(2⦃_,/ _⦄)")
```

translations
```
  "{|x, y, z|}"   == "{|x, {|y, z|}|}"
  "{|x, y|}"      == "MPair x y"
```

constdefs
```
  nat_of_agent :: "agent => nat"
   "nat_of_agent == agent_case (curry nat2_to_nat 0)
                               (curry nat2_to_nat 1)
                               (curry nat2_to_nat 2)
                               (curry nat2_to_nat 3)
                               (nat2_to_nat (4,0))"
```
— maps each agent to a unique natural number, for specifications

The function is indeed injective

lemma *inj_nat_of_agent: "inj nat_of_agent"*
by *(simp add: nat_of_agent_def inj_on_def curry_def nat2_to_nat_inj [THEN inj_eq] split: agent.split)*

constdefs
```
  keysFor :: "msg set => key set"
  "keysFor H == invKey ` {K. ∃X. Crypt K X ∈ H}"
```

1.1.1 Inductive definition of all "parts" of a message.

consts *parts :: "msg set => msg set"*
inductive *"parts H"*
intros
```
    Inj [intro]:              "X ∈ H ==> X ∈ parts H"
    Fst:         "{|X,Y|}   ∈ parts H ==> X ∈ parts H"
    Snd:         "{|X,Y|}   ∈ parts H ==> Y ∈ parts H"
    Body:        "Crypt K X ∈ parts H ==> X ∈ parts H"
```

lemma *parts_mono: "G<=H ==> parts(G) <= parts(H)"*
apply *auto*
apply *(erule parts.induct)*

apply *(auto dest: Fst Snd Body)*
done

1.1.2 Inverse of keys

lemma *Key_image_eq [simp]: "(Key x ∈ Key'A) = (x∈A)"*
by *auto*

lemma *Nonce_Key_image_eq [simp]: "(Nonce x ∉ Key'A)"*
by *auto*

lemma *Cardholder_image_eq [simp]: "(Cardholder x ∈ Cardholder'A) = (x ∈ A)"*
by *auto*

lemma *CA_image_eq [simp]: "(CA x ∈ CA'A) = (x ∈ A)"*
by *auto*

lemma *Pan_image_eq [simp]: "(Pan x ∈ Pan'A) = (x ∈ A)"*
by *auto*

lemma *Pan_Key_image_eq [simp]: "(Pan x ∉ Key'A)"*
by *auto*

lemma *Nonce_Pan_image_eq [simp]: "(Nonce x ∉ Pan'A)"*
by *auto*

lemma *invKey_eq [simp]: "(invKey K = invKey K') = (K=K')"*
apply *safe*
apply *(drule_tac f = invKey* **in** *arg_cong, simp)*
done

1.2 keysFor operator

lemma *keysFor_empty [simp]: "keysFor {} = {}"*
by *(unfold keysFor_def, blast)*

lemma *keysFor_Un [simp]: "keysFor (H ∪ H') = keysFor H ∪ keysFor H'"*
by *(unfold keysFor_def, blast)*

lemma *keysFor_UN [simp]: "keysFor (⋃i∈A. H i) = (⋃i∈A. keysFor (H i))"*
by *(unfold keysFor_def, blast)*

lemma *keysFor_mono: "G⊆H ==> keysFor(G) ⊆ keysFor(H)"*
by *(unfold keysFor_def, blast)*

lemma *keysFor_insert_Agent [simp]: "keysFor (insert (Agent A) H) = keysFor H"*
by *(unfold keysFor_def, auto)*

lemma *keysFor_insert_Nonce [simp]: "keysFor (insert (Nonce N) H) = keysFor H"*
by *(unfold keysFor_def, auto)*

```
lemma keysFor_insert_Number [simp]: "keysFor (insert (Number N) H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Key [simp]: "keysFor (insert (Key K) H) = keysFor H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Pan [simp]: "keysFor (insert (Pan A) H) = keysFor H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Hash [simp]: "keysFor (insert (Hash X) H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_MPair [simp]: "keysFor (insert {|X,Y|} H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Crypt [simp]:
    "keysFor (insert (Crypt K X) H) = insert (invKey K) (keysFor H)"
by (unfold keysFor_def, auto)

lemma keysFor_image_Key [simp]: "keysFor (Key`E) = {}"
by (unfold keysFor_def, auto)

lemma Crypt_imp_invKey_keysFor: "Crypt K X ∈ H ==> invKey K ∈ keysFor H"
by (unfold keysFor_def, blast)
```

1.3 Inductive relation "parts"

```
lemma MPair_parts:
     "[| {|X,Y|} ∈ parts H;
         [| X ∈ parts H; Y ∈ parts H |] ==> P |] ==> P"
by (blast dest: parts.Fst parts.Snd)

declare MPair_parts [elim!]  parts.Body [dest!]
```

NB These two rules are UNSAFE in the formal sense, as they discard the compound message. They work well on THIS FILE. *MPair_parts* is left as SAFE because it speeds up proofs. The Crypt rule is normally kept UNSAFE to avoid breaking up certificates.

```
lemma parts_increasing: "H ⊆ parts(H)"
by blast

lemmas parts_insertI = subset_insertI [THEN parts_mono, THEN subsetD, standard]

lemma parts_empty [simp]: "parts{} = {}"
apply safe
apply (erule parts.induct, blast+)
done

lemma parts_emptyE [elim!]: "X∈ parts{} ==> P"
by simp
```

lemma *parts_singleton: "X∈ parts H ==> ∃Y∈H. X∈ parts {Y}"*
by *(erule parts.induct, blast+)*

1.3.1 Unions

lemma *parts_Un_subset1: "parts(G) ∪ parts(H) ⊆ parts(G ∪ H)"*
by *(intro Un_least parts_mono Un_upper1 Un_upper2)*

lemma *parts_Un_subset2: "parts(G ∪ H) ⊆ parts(G) ∪ parts(H)"*
apply *(rule subsetI)*
apply *(erule parts.induct, blast+)*
done

lemma *parts_Un [simp]: "parts(G ∪ H) = parts(G) ∪ parts(H)"*
by *(intro equalityI parts_Un_subset1 parts_Un_subset2)*

lemma *parts_insert: "parts (insert X H) = parts {X} ∪ parts H"*
apply *(subst insert_is_Un [of _ H])*
apply *(simp only: parts_Un)*
done

lemma *parts_insert2:*
"parts (insert X (insert Y H)) = parts {X} ∪ parts {Y} ∪ parts H"
apply *(simp add: Un_assoc)*
apply *(simp add: parts_insert [symmetric])*
done

lemma *parts_UN_subset1: "(⋃x∈A. parts(H x)) ⊆ parts(⋃x∈A. H x)"*
by *(intro UN_least parts_mono UN_upper)*

lemma *parts_UN_subset2: "parts(⋃x∈A. H x) ⊆ (⋃x∈A. parts(H x))"*
apply *(rule subsetI)*
apply *(erule parts.induct, blast+)*
done

lemma *parts_UN [simp]: "parts(⋃x∈A. H x) = (⋃x∈A. parts(H x))"*
by *(intro equalityI parts_UN_subset1 parts_UN_subset2)*

This allows `blast` to simplify occurrences of *parts (G ∪ H)* in the assumption.

declare *parts_Un [THEN equalityD1, THEN subsetD, THEN UnE, elim!]*

lemma *parts_insert_subset: "insert X (parts H) ⊆ parts(insert X H)"*
by *(blast intro: parts_mono [THEN [2] rev_subsetD])*

1.3.2 Idempotence and transitivity

lemma *parts_partsD [dest!]: "X∈ parts (parts H) ==> X∈ parts H"*
by *(erule parts.induct, blast+)*

lemma *parts_idem [simp]: "parts (parts H) = parts H"*

```
by blast

lemma parts_trans: "[| X∈ parts G;  G ⊆ parts H |] ==> X∈ parts H"
by (drule parts_mono, blast)


lemma parts_cut:
     "[| Y∈ parts (insert X G);  X∈ parts H |] ==> Y∈ parts (G ∪ H)"
by (erule parts_trans, auto)

lemma parts_cut_eq [simp]: "X∈ parts H ==> parts (insert X H) = parts H"
by (force dest!: parts_cut intro: parts_insertI)
```

1.3.3 Rewrite rules for pulling out atomic messages

```
lemmas parts_insert_eq_I = equalityI [OF subsetI parts_insert_subset]


lemma parts_insert_Agent [simp]:
     "parts (insert (Agent agt) H) = insert (Agent agt) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done

lemma parts_insert_Nonce [simp]:
     "parts (insert (Nonce N) H) = insert (Nonce N) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done

lemma parts_insert_Number [simp]:
     "parts (insert (Number N) H) = insert (Number N) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done

lemma parts_insert_Key [simp]:
     "parts (insert (Key K) H) = insert (Key K) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done

lemma parts_insert_Pan [simp]:
     "parts (insert (Pan A) H) = insert (Pan A) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done

lemma parts_insert_Hash [simp]:
     "parts (insert (Hash X) H) = insert (Hash X) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done
```

```
lemma parts_insert_Crypt [simp]:
     "parts (insert (Crypt K X) H) =
          insert (Crypt K X) (parts (insert X H))"
apply (rule equalityI)
apply (rule subsetI)
apply (erule parts.induct, auto)
apply (erule parts.induct)
apply (blast intro: parts.Body)+
done

lemma parts_insert_MPair [simp]:
     "parts (insert {|X,Y|} H) =
          insert {|X,Y|} (parts (insert X (insert Y H)))"
apply (rule equalityI)
apply (rule subsetI)
apply (erule parts.induct, auto)
apply (erule parts.induct)
apply (blast intro: parts.Fst parts.Snd)+
done

lemma parts_image_Key [simp]: "parts (Key`N) = Key`N"
apply auto
apply (erule parts.induct, auto)
done

lemma parts_image_Pan [simp]: "parts (Pan`A) = Pan`A"
apply auto
apply (erule parts.induct, auto)
done



lemma msg_Nonce_supply: "∃N. ∀n. N≤n --> Nonce n ∉ parts {msg}"
apply (induct_tac "msg")
apply (simp_all (no_asm_simp) add: exI parts_insert2)

prefer 2 apply (blast elim!: add_leE)

apply (rule_tac x = "N + Suc nat" in exI)
apply (auto elim!: add_leE)
done


lemma msg_Number_supply: "∃N. ∀n. N<=n --> Number n ∉ parts {msg}"
apply (induct_tac "msg")
apply (simp_all (no_asm_simp) add: exI parts_insert2)
prefer 2 apply (blast elim!: add_leE)
apply (rule_tac x = "N + Suc nat" in exI, auto)
done
```

1.4 Inductive relation "analz"

Inductive definition of "analz" – what can be broken down from a set of messages, including keys. A form of downward closure. Pairs can be taken apart;

messages decrypted with known keys.

```
consts  analz   :: "msg set => msg set"
inductive "analz H"
  intros
    Inj [intro,simp] :    "X ∈ H ==> X ∈ analz H"
    Fst:     "{|X,Y|} ∈ analz H ==> X ∈ analz H"
    Snd:     "{|X,Y|} ∈ analz H ==> Y ∈ analz H"
    Decrypt [dest]:
             "[|Crypt K X ∈ analz H; Key(invKey K): analz H|] ==> X ∈ analz
H"
```

```
lemma analz_mono: "G<=H ==> analz(G) <= analz(H)"
apply auto
apply (erule analz.induct)
apply (auto dest: Fst Snd)
done
```

Making it safe speeds up proofs

```
lemma MPair_analz [elim!]:
     "[| {|X,Y|} ∈ analz H;
             [| X ∈ analz H; Y ∈ analz H |] ==> P
          |] ==> P"
by (blast dest: analz.Fst analz.Snd)

lemma analz_increasing: "H ⊆ analz(H)"
by blast

lemma analz_subset_parts: "analz H ⊆ parts H"
apply (rule subsetI)
apply (erule analz.induct, blast+)
done

lemmas analz_into_parts = analz_subset_parts [THEN subsetD, standard]

lemmas not_parts_not_analz = analz_subset_parts [THEN contra_subsetD, standard]


lemma parts_analz [simp]: "parts (analz H) = parts H"
apply (rule equalityI)
apply (rule analz_subset_parts [THEN parts_mono, THEN subset_trans], simp)
apply (blast intro: analz_increasing [THEN parts_mono, THEN subsetD])
done

lemma analz_parts [simp]: "analz (parts H) = parts H"
apply auto
apply (erule analz.induct, auto)
done

lemmas analz_insertI = subset_insertI [THEN analz_mono, THEN [2] rev_subsetD,
standard]
```

1.4.1 General equational properties

```
lemma analz_empty [simp]: "analz{} = {}"
apply safe
apply (erule analz.induct, blast+)
done


lemma analz_Un: "analz(G) ∪ analz(H) ⊆ analz(G ∪ H)"
by (intro Un_least analz_mono Un_upper1 Un_upper2)

lemma analz_insert: "insert X (analz H) ⊆ analz(insert X H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])
```

1.4.2 Rewrite rules for pulling out atomic messages

```
lemmas analz_insert_eq_I = equalityI [OF subsetI analz_insert]

lemma analz_insert_Agent [simp]:
     "analz (insert (Agent agt) H) = insert (Agent agt) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

lemma analz_insert_Nonce [simp]:
     "analz (insert (Nonce N) H) = insert (Nonce N) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

lemma analz_insert_Number [simp]:
     "analz (insert (Number N) H) = insert (Number N) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

lemma analz_insert_Hash [simp]:
     "analz (insert (Hash X) H) = insert (Hash X) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done


lemma analz_insert_Key [simp]:
    "K ∉ keysFor (analz H) ==>
          analz (insert (Key K) H) = insert (Key K) (analz H)"
apply (unfold keysFor_def)
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

lemma analz_insert_MPair [simp]:
     "analz (insert {|X,Y|} H) =
          insert {|X,Y|} (analz (insert X (insert Y H)))"
apply (rule equalityI)
```

```
apply (rule subsetI)
apply (erule analz.induct, auto)
apply (erule analz.induct)
apply (blast intro: analz.Fst analz.Snd)+
done


lemma analz_insert_Crypt:
     "Key (invKey K) ∉ analz H
      ==> analz (insert (Crypt K X) H) = insert (Crypt K X) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

lemma analz_insert_Pan [simp]:
     "analz (insert (Pan A) H) = insert (Pan A) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

lemma lemma1: "Key (invKey K) ∈ analz H ==>
               analz (insert (Crypt K X) H) ⊆
               insert (Crypt K X) (analz (insert X H))"
apply (rule subsetI)
apply (erule_tac xa = x in analz.induct, auto)
done

lemma lemma2: "Key (invKey K) ∈ analz H ==>
               insert (Crypt K X) (analz (insert X H)) ⊆
               analz (insert (Crypt K X) H)"
apply auto
apply (erule_tac xa = x in analz.induct, auto)
apply (blast intro: analz_insertI analz.Decrypt)
done

lemma analz_insert_Decrypt:
     "Key (invKey K) ∈ analz H ==>
               analz (insert (Crypt K X) H) =
               insert (Crypt K X) (analz (insert X H))"
by (intro equalityI lemma1 lemma2)


lemma analz_Crypt_if [simp]:
     "analz (insert (Crypt K X) H) =
          (if (Key (invKey K) ∈ analz H)
           then insert (Crypt K X) (analz (insert X H))
           else insert (Crypt K X) (analz H))"
by (simp add: analz_insert_Crypt analz_insert_Decrypt)



lemma analz_insert_Crypt_subset:
     "analz (insert (Crypt K X) H) ⊆
           insert (Crypt K X) (analz (insert X H))"
```

apply *(rule subsetI)*
apply *(erule analz.induct, auto)*
done

lemma *analz_image_Key [simp]: "analz (Key‘N) = Key‘N"*
apply *auto*
apply *(erule analz.induct, auto)*
done

lemma *analz_image_Pan [simp]: "analz (Pan‘A) = Pan‘A"*
apply *auto*
apply *(erule analz.induct, auto)*
done

1.4.3 Idempotence and transitivity

lemma *analz_analzD [dest!]: "X∈ analz (analz H) ==> X∈ analz H"*
by *(erule analz.induct, blast+)*

lemma *analz_idem [simp]: "analz (analz H) = analz H"*
by *blast*

lemma *analz_trans: "[| X∈ analz G; G ⊆ analz H |] ==> X∈ analz H"*
by *(drule analz_mono, blast)*

lemma *analz_cut: "[| Y∈ analz (insert X H); X∈ analz H |] ==> Y∈ analz H"*
by *(erule analz_trans, blast)*

lemma *analz_insert_eq: "X∈ analz H ==> analz (insert X H) = analz H"*
by *(blast intro: analz_cut analz_insertI)*

A congruence rule for "analz"

lemma *analz_subset_cong:*
    "[| analz G ⊆ analz G'; analz H ⊆ analz H'
        |] ==> analz (G ∪ H) ⊆ analz (G' ∪ H')"
apply *clarify*
apply *(erule analz.induct)*
apply *(best intro: analz_mono [THEN subsetD])+*
done

lemma *analz_cong:*
    "[| analz G = analz G'; analz H = analz H'
        |] ==> analz (G ∪ H) = analz (G' ∪ H')"
by *(intro equalityI analz_subset_cong, simp_all)*

lemma *analz_insert_cong:*
    "analz H = analz H' ==> analz(insert X H) = analz(insert X H')"
by *(force simp only: insert_def intro!: analz_cong)*

```
lemma analz_trivial:
     "[| ∀X Y. {|X,Y|} ∉ H;  ∀X K. Crypt K X ∉ H |] ==> analz H = H"
apply safe
apply (erule analz.induct, blast+)
done


lemma analz_UN_analz_lemma:
     "X∈ analz (⋃i∈A. analz (H i)) ==> X∈ analz (⋃i∈A. H i)"
apply (erule analz.induct)
apply (blast intro: analz_mono [THEN [2] rev_subsetD])+
done

lemma analz_UN_analz [simp]: "analz (⋃i∈A. analz (H i)) = analz (⋃i∈A.
H i)"
by (blast intro: analz_UN_analz_lemma analz_mono [THEN [2] rev_subsetD])
```

1.5 Inductive relation "synth"

Inductive definition of "synth" – what can be built up from a set of messages. A form of upward closure. Pairs can be built, messages encrypted with known keys. Agent names are public domain. Numbers can be guessed, but Nonces cannot be.

```
consts  synth   :: "msg set => msg set"
inductive "synth H"
  intros
    Inj    [intro]:   "X ∈ H ==> X ∈ synth H"
    Agent  [intro]:   "Agent agt ∈ synth H"
    Number [intro]:   "Number n  ∈ synth H"
    Hash   [intro]:   "X ∈ synth H ==> Hash X ∈ synth H"
    MPair  [intro]:   "[|X ∈ synth H;  Y ∈ synth H|] ==> {|X,Y|} ∈ synth
H"
    Crypt  [intro]:   "[|X ∈ synth H;  Key(K) ∈ H|] ==> Crypt K X ∈ synth
H"


lemma synth_mono: "G<=H ==> synth(G) <= synth(H)"
apply auto
apply (erule synth.induct)
apply (auto dest: Fst Snd Body)
done


inductive_cases Nonce_synth [elim!]: "Nonce n ∈ synth H"
inductive_cases Key_synth   [elim!]: "Key K ∈ synth H"
inductive_cases Hash_synth  [elim!]: "Hash X ∈ synth H"
inductive_cases MPair_synth [elim!]: "{|X,Y|} ∈ synth H"
inductive_cases Crypt_synth [elim!]: "Crypt K X ∈ synth H"
inductive_cases Pan_synth   [elim!]: "Pan A ∈ synth H"


lemma synth_increasing: "H ⊆ synth(H)"
```

by *blast*

1.5.1 Unions

lemma *synth_Un: "synth(G) ∪ synth(H) ⊆ synth(G ∪ H)"*
by *(intro Un_least synth_mono Un_upper1 Un_upper2)*

lemma *synth_insert: "insert X (synth H) ⊆ synth(insert X H)"*
by *(blast intro: synth_mono [THEN [2] rev_subsetD])*

1.5.2 Idempotence and transitivity

lemma *synth_synthD [dest!]: "X∈ synth (synth H) ==> X∈ synth H"*
by *(erule synth.induct, blast+)*

lemma *synth_idem: "synth (synth H) = synth H"*
by *blast*

lemma *synth_trans: "[| X∈ synth G; G ⊆ synth H |] ==> X∈ synth H"*
by *(drule synth_mono, blast)*

lemma *synth_cut: "[| Y∈ synth (insert X H); X∈ synth H |] ==> Y∈ synth H"*
by *(erule synth_trans, blast)*

lemma *Agent_synth [simp]: "Agent A ∈ synth H"*
by *blast*

lemma *Number_synth [simp]: "Number n ∈ synth H"*
by *blast*

lemma *Nonce_synth_eq [simp]: "(Nonce N ∈ synth H) = (Nonce N ∈ H)"*
by *blast*

lemma *Key_synth_eq [simp]: "(Key K ∈ synth H) = (Key K ∈ H)"*
by *blast*

lemma *Crypt_synth_eq [simp]: "Key K ∉ H ==> (Crypt K X ∈ synth H) = (Crypt K X ∈ H)"*
by *blast*

lemma *Pan_synth_eq [simp]: "(Pan A ∈ synth H) = (Pan A ∈ H)"*
by *blast*

lemma *keysFor_synth [simp]:*
"keysFor (synth H) = keysFor H ∪ invKey‘{K. Key K ∈ H}"
by *(unfold keysFor_def, blast)*

1.5.3 Combinations of parts, analz and synth

lemma *parts_synth [simp]: "parts (synth H) = parts H ∪ synth H"*
apply *(rule equalityI)*
apply *(rule subsetI)*
apply *(erule parts.induct)*

```
apply (blast intro: synth_increasing [THEN parts_mono, THEN subsetD]
                    parts.Fst parts.Snd parts.Body)+
done

lemma analz_analz_Un [simp]: "analz (analz G ∪ H) = analz (G ∪ H)"
apply (intro equalityI analz_subset_cong)+
apply simp_all
done

lemma analz_synth_Un [simp]: "analz (synth G ∪ H) = analz (G ∪ H) ∪ synth
G"
apply (rule equalityI)
apply (rule subsetI)
apply (erule analz.induct)
prefer 5 apply (blast intro: analz_mono [THEN [2] rev_subsetD])
apply (blast intro: analz.Fst analz.Snd analz.Decrypt)+
done

lemma analz_synth [simp]: "analz (synth H) = analz H ∪ synth H"
apply (cut_tac H = "{}" in analz_synth_Un)
apply (simp (no_asm_use))
done
```

1.5.4 For reasoning about the Fake rule in traces

```
lemma parts_insert_subset_Un: "X∈ G ==> parts(insert X H) ⊆ parts G ∪ parts
H"
by (rule subset_trans [OF parts_mono parts_Un_subset2], blast)


lemma Fake_parts_insert: "X ∈ synth (analz H) ==>
      parts (insert X H) ⊆ synth (analz H) ∪ parts H"
apply (drule parts_insert_subset_Un)
apply (simp (no_asm_use))
apply blast
done

lemma Fake_parts_insert_in_Un:
     "[|Z ∈ parts (insert X H);  X: synth (analz H)|]
      ==> Z ∈  synth (analz H) ∪ parts H"
by (blast dest: Fake_parts_insert [THEN subsetD, dest])


lemma Fake_analz_insert:
     "X∈ synth (analz G) ==>
      analz (insert X H) ⊆ synth (analz G) ∪ analz (G ∪ H)"
apply (rule subsetI)
apply (subgoal_tac "x ∈ analz (synth (analz G) ∪ H) ")
prefer 2 apply (blast intro: analz_mono [THEN [2] rev_subsetD] analz_mono
[THEN synth_mono, THEN [2] rev_subsetD])
apply (simp (no_asm_use))
apply blast
done
```

```
lemma analz_conj_parts [simp]:
     "(X ∈ analz H & X ∈ parts H) = (X ∈ analz H)"
by (blast intro: analz_subset_parts [THEN subsetD])

lemma analz_disj_parts [simp]:
     "(X ∈ analz H | X ∈ parts H) = (X ∈ parts H)"
by (blast intro: analz_subset_parts [THEN subsetD])


lemma MPair_synth_analz [iff]:
     "({|X,Y|} ∈ synth (analz H)) =
      (X ∈ synth (analz H) & Y ∈ synth (analz H))"
by blast

lemma Crypt_synth_analz:
     "[| Key K ∈ analz H;  Key (invKey K) ∈ analz H |]
       ==> (Crypt K X ∈ synth (analz H)) = (X ∈ synth (analz H))"
by blast


lemma Hash_synth_analz [simp]:
     "X ∉ synth (analz H)
      ==> (Hash{|X,Y|} ∈ synth (analz H)) = (Hash{|X,Y|} ∈ analz H)"
by blast



declare parts.Body [rule del]
```

Rewrites to push in Key and Crypt messages, so that other messages can be pulled out using the `analz_insert` rules

```
ML
{*
fun insComm x y = inst "x" x (inst "y" y insert_commute);

bind_thms ("pushKeys",
           map (insComm "Key ?K")
                   ["Agent ?C", "Nonce ?N", "Number ?N", "Pan ?PAN",
                    "Hash ?X", "MPair ?X ?Y", "Crypt ?X ?K'"]);

bind_thms ("pushCrypts",
           map (insComm "Crypt ?X ?K")
                     ["Agent ?C", "Nonce ?N", "Number ?N", "Pan ?PAN",
                      "Hash ?X'", "MPair ?X' ?Y"]);
*}
```

Cannot be added with `[simp]` – messages should not always be re-ordered.

```
lemmas pushes = pushKeys pushCrypts
```

1.6 Tactics useful for many protocol proofs

```
declare o_def [simp]


lemma Crypt_notin_image_Key [simp]: "Crypt K X ∉ Key ` A"
by auto

lemma Hash_notin_image_Key [simp] :"Hash X ∉ Key ` A"
by auto

lemma synth_analz_mono: "G<=H ==> synth (analz(G)) <= synth (analz(H))"
by (simp add: synth_mono analz_mono)

lemma Fake_analz_eq [simp]:
     "X ∈ synth(analz H) ==> synth (analz (insert X H)) = synth (analz H)"
apply (drule Fake_analz_insert[of _ _ "H"])
apply (simp add: synth_increasing[THEN Un_absorb2])
apply (drule synth_mono)
apply (simp add: synth_idem)
apply (blast intro: synth_analz_mono [THEN [2] rev_subsetD])
done
```

Two generalizations of `analz_insert_eq`

```
lemma gen_analz_insert_eq [rule_format]:
     "X ∈ analz H ==> ALL G. H ⊆ G --> analz (insert X G) = analz G"
by (blast intro: analz_cut analz_insertI analz_mono [THEN [2] rev_subsetD])

lemma synth_analz_insert_eq [rule_format]:
     "X ∈ synth (analz H)
      ==> ALL G. H ⊆ G --> (Key K ∈ analz (insert X G)) = (Key K ∈ analz
G)"
apply (erule synth.induct)
apply (simp_all add: gen_analz_insert_eq subset_trans [OF _ subset_insertI])
done

lemma Fake_parts_sing:
     "X ∈ synth (analz H) ==> parts{X} ⊆ synth (analz H) ∪ parts H"
apply (rule subset_trans)
 apply (erule_tac [2] Fake_parts_insert)
apply (simp add: parts_mono)
done

lemmas Fake_parts_sing_imp_Un = Fake_parts_sing [THEN [2] rev_subsetD]

method_setup spy_analz = {*
    Method.ctxt_args (fn ctxt =>
        Method.METHOD (fn facts =>
            gen_spy_analz_tac (Classical.get_local_claset ctxt,
                               Simplifier.get_local_simpset ctxt) 1))*}
    "for proving the Fake case when analz is involved"

method_setup atomic_spy_analz = {*
    Method.ctxt_args (fn ctxt =>
        Method.METHOD (fn facts =>
```

```
            atomic_spy_analz_tac (Classical.get_local_claset ctxt,
                                  Simplifier.get_local_simpset ctxt) 1))*}
    "for debugging spy_analz"
```

method_setup *Fake_insert_simp = {**

```
    Method.ctxt_args (fn ctxt =>
        Method.METHOD (fn facts =>
            Fake_insert_simp_tac (Simplifier.get_local_simpset ctxt) 1))*}
    "for debugging spy_analz"
```

end

2 Theory of Events for SET

theory *EventSET = MessageSET:*

The Root Certification Authority

syntax *RCA :: agent*
translations *"RCA" == "CA 0"*

Message events

datatype

```
  event = Says  agent agent msg
        | Gets  agent       msg
        | Notes agent       msg
```

compromised agents: keys known, Notes visible

consts *bad :: "agent set"*

Spy has access to his own key for spoof messages, but RCA is secure

specification *(bad)*

```
  Spy_in_bad     [iff]: "Spy ∈ bad"
  RCA_not_bad [iff]: "RCA ∉ bad"
    by (rule exI [of _ "{Spy}"], simp)
```

2.1 Agents' Knowledge

consts

```
  initState :: "agent => msg set"
  knows  :: "[agent, event list] => msg set"
```

primrec

```
knows_Nil:
  "knows A []       = initState A"
knows_Cons:
    "knows A (ev # evs) =
       (if A = Spy then
        (case ev of
```

```
     Says A' B X => insert X (knows Spy evs)
    | Gets A' X => knows Spy evs
    | Notes A' X  =>
       if A' ∈ bad then insert X (knows Spy evs) else knows Spy evs)
  else
  (case ev of
     Says A' B X =>
       if A'=A then insert X (knows A evs) else knows A evs
    | Gets A' X    =>
       if A'=A then insert X (knows A evs) else knows A evs
    | Notes A' X    =>
       if A'=A then insert X (knows A evs) else knows A evs))"
```

2.2 Used Messages

consts

```
  used :: "event list => msg set"
```

primrec

```
  used_Nil:  "used []         = (UN B. parts (initState B))"
  used_Cons: "used (ev # evs) =
                 (case ev of
                    Says A B X => parts {X} Un (used evs)
                  | Gets A X   => used evs
                  | Notes A X  => parts {X} Un (used evs))"
```

lemmas *parts_insert_knows_A = parts_insert [of _ "knows A evs", standard]*

lemma *knows_Spy_Says [simp]:*
"knows Spy (Says A B X # evs) = insert X (knows Spy evs)"
by *auto*

Letting the Spy see "bad" agents' notes avoids redundant case-splits on whether *A = Spy* and whether *A ∈ bad*

lemma *knows_Spy_Notes [simp]:*
"knows Spy (Notes A X # evs) =
(if A:bad then insert X (knows Spy evs) else knows Spy evs)"
apply *auto*
done

lemma *knows_Spy_Gets [simp]: "knows Spy (Gets A X # evs) = knows Spy evs"*
by *auto*

lemma *initState_subset_knows: "initState A <= knows A evs"*
apply *(induct_tac "evs")*
apply *(auto split: event.split)*

done

lemma *knows_Spy_subset_knows_Spy_Says:*
"knows Spy evs <= knows Spy (Says A B X # evs)"
by *auto*

lemma *knows_Spy_subset_knows_Spy_Notes:*
"knows Spy evs <= knows Spy (Notes A X # evs)"
by *auto*

lemma *knows_Spy_subset_knows_Spy_Gets:*
"knows Spy evs <= knows Spy (Gets A X # evs)"
by *auto*

lemma *Says_imp_knows_Spy [rule_format]:*
"Says A B X ∈ set evs --> X ∈ knows Spy evs"
apply *(induct_tac "evs")*
apply *(auto split: event.split)*
done

lemmas *knows_Spy_partsEs =*
Says_imp_knows_Spy [THEN parts.Inj, THEN revcut_rl, standard]
parts.Body [THEN revcut_rl, standard]

2.3 The Function *used*

lemma *parts_knows_Spy_subset_used: "parts (knows Spy evs) <= used evs"*
apply *(induct_tac "evs")*
apply *(auto simp add: parts_insert_knows_A split: event.split)*
done

lemmas *usedI = parts_knows_Spy_subset_used [THEN subsetD, intro]*

lemma *initState_subset_used: "parts (initState B) <= used evs"*
apply *(induct_tac "evs")*
apply *(auto split: event.split)*
done

lemmas *initState_into_used = initState_subset_used [THEN subsetD]*

lemma *used_Says [simp]: "used (Says A B X # evs) = parts{X} Un used evs"*
by *auto*

lemma *used_Notes [simp]: "used (Notes A X # evs) = parts{X} Un used evs"*
by *auto*

lemma *used_Gets [simp]: "used (Gets A X # evs) = used evs"*
by *auto*

lemma *Notes_imp_parts_subset_used [rule_format]:*
"Notes A X ∈ set evs --> parts {X} <= used evs"

```
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done
```

NOTE REMOVAL–laws above are cleaner, as they don't involve "case"

```
declare knows_Cons [simp del]
        used_Nil [simp del] used_Cons [simp del]
```

For proving theorems of the form `X ∉ analz (knows Spy evs) ⟶ P` New events added by induction to "evs" are discarded. Provided this information isn't needed, the proof will be much shorter, since it will omit complicated reasoning about `analz`.

```
lemmas analz_mono_contra =
       knows_Spy_subset_knows_Spy_Says [THEN analz_mono, THEN contra_subsetD]
       knows_Spy_subset_knows_Spy_Notes [THEN analz_mono, THEN contra_subsetD]
       knows_Spy_subset_knows_Spy_Gets [THEN analz_mono, THEN contra_subsetD]
ML
{*
val analz_mono_contra_tac =
  let val analz_impI = inst "P" "?Y ∉ analz (knows Spy ?evs)" impI
  in rtac analz_impI THEN'
     REPEAT1 o (dresolve_tac (thms"analz_mono_contra")) THEN'
     mp_tac
  end
*}

method_setup analz_mono_contra = {*
    Method.no_args
      (Method.METHOD (fn facts => REPEAT_FIRST analz_mono_contra_tac)) *}
    "for proving theorems of the form X ∉ analz (knows Spy evs) --> P"

end
```

3 The Public-Key Theory, Modified for SET

```
theory PublicSET = EventSET:
```

3.1 Symmetric and Asymmetric Keys

definitions influenced by the wish to assign asymmetric keys - since the beginning - only to RCA and CAs, namely we need a partial function on type Agent

The SET specs mention two signature keys for CAs - we only have one

```
consts
  publicKey :: "[bool, agent] => key"
```
— the boolean is TRUE if a signing key

```
syntax
  pubEK :: "agent => key"
  pubSK :: "agent => key"
  priEK :: "agent => key"
```

```
  priSK :: "agent => key"
```

translations
```
  "pubEK"  == "publicKey False"
  "pubSK"  == "publicKey True"


  "priEK A"  == "invKey (pubEK A)"
  "priSK A"  == "invKey (pubSK A)"
```

By freeness of agents, no two agents have the same key. Since `True` ≠ `False`, no agent has the same signing and encryption keys.

specification `(publicKey)`
```
  injective_publicKey:
    "publicKey b A = publicKey c A' ==> b=c & A=A'"
```
axioms
```
  privateKey_neq_publicKey [iff]:
      "invKey (publicKey b A) ≠ publicKey b' A'"
```

declare `privateKey_neq_publicKey [THEN not_sym, iff]`

3.2 Initial Knowledge

This information is not necessary. Each protocol distributes any needed certificates, and anyway our proofs require a formalization of the Spy's knowledge only. However, the initial knowledge is as follows: All agents know RCA's public keys; RCA and CAs know their own respective keys; RCA (has already certified and therefore) knows all CAs public keys; Spy knows all keys of all bad agents.

primrec
```
  initState_Spy:
    "initState Spy = Key ` (invKey ` pubEK ` bad Un
                            invKey ` pubSK ` bad Un
                            range pubEK Un range pubSK)"
```

Injective mapping from agents to PANs: an agent can have only one card

consts `pan :: "agent => nat"`

specification `(pan)`
```
  inj_pan: "inj pan"
```
— No two agents have the same PAN
declare `inj_pan [THEN inj_eq, iff]`

consts
`XOR :: "nat*nat => nat"` — no properties are assumed of exclusive-or

3.3 Signature Primitives

constdefs

```
  sign :: "[key, msg]=>msg"
```

```
    "sign K X == {|X, Crypt K (Hash X) |}"


  signOnly :: "[key, msg]=>msg"
    "signOnly K X == Crypt K (Hash X)"


  signCert :: "[key, msg]=>msg"
    "signCert K X == {|X, Crypt K X |}"


  cert :: "[agent, key, msg, key] => msg"
    "cert A Ka T signK == signCert signK {|Agent A, Key Ka, T|}"



  certC :: "[nat, key, nat, msg, key] => msg"
    "certC PAN Ka PS T signK ==
     signCert signK {|Hash {|Nonce PS, Pan PAN|}, Key Ka, T|}"
```

syntax

```
  "onlyEnc" :: msg
  "onlySig" :: msg
  "authCode" :: msg
```

translations

```
  "onlyEnc"   == "Number 0"
  "onlySig"  == "Number (Suc 0)"
  "authCode" == "Number (Suc (Suc 0))"
```

3.4 Encryption Primitives

constdefs

```
  EXcrypt :: "[key,key,msg,msg] => msg"
```
— Extra Encryption

```
    "EXcrypt K EK M m ==
       {|Crypt K {|M, Hash m|}, Crypt EK {|Key K, m|}|}"

  EXHcrypt :: "[key,key,msg,msg] => msg"
```
— Extra Encryption with Hashing

```
    "EXHcrypt K EK M m ==
       {|Crypt K {|M, Hash m|}, Crypt EK {|Key K, m, Hash M|}|}"

  Enc :: "[key,key,key,msg] => msg"
```
— Simple Encapsulation with SIGNATURE

```
    "Enc SK K EK M ==
       {|Crypt K (sign SK M), Crypt EK (Key K)|}"
```

```
EncB :: "[key,key,key,msg,msg] => msg"
```
— Encapsulation with Baggage. Keys as above, and baggage b.
```
  "EncB SK K EK M b ==
     {|Enc SK K EK {|M, Hash b|}, b|}"
```

3.5 Basic Properties of pubEK, pubSK, priEK and priSK

```
lemma publicKey_eq_iff [iff]:
     "(publicKey b A = publicKey b' A') = (b=b' & A=A')"
by (blast dest: injective_publicKey)

lemma privateKey_eq_iff [iff]:
     "(invKey (publicKey b A) = invKey (publicKey b' A')) = (b=b' & A=A')"
by auto

lemma not_symKeys_publicKey [iff]: "publicKey b A ∉ symKeys"
by (simp add: symKeys_def)

lemma not_symKeys_privateKey [iff]: "invKey (publicKey b A) ∉ symKeys"
by (simp add: symKeys_def)

lemma symKeys_invKey_eq [simp]: "K ∈ symKeys ==> invKey K = K"
by (simp add: symKeys_def)

lemma symKeys_invKey_iff [simp]: "(invKey K ∈ symKeys) = (K ∈ symKeys)"
by (unfold symKeys_def, auto)
```

Can be slow (or even loop) as a simprule

```
lemma symKeys_neq_imp_neq: "(K ∈ symKeys) ≠ (K' ∈ symKeys) ==> K ≠ K'"
by blast
```

These alternatives to *symKeys_neq_imp_neq* don't seem any better in practice.

```
lemma publicKey_neq_symKey: "K ∈ symKeys ==> publicKey b A ≠ K"
by blast

lemma symKey_neq_publicKey: "K ∈ symKeys ==> K ≠ publicKey b A"
by blast

lemma privateKey_neq_symKey: "K ∈ symKeys ==> invKey (publicKey b A) ≠ K"
by blast

lemma symKey_neq_privateKey: "K ∈ symKeys ==> K ≠ invKey (publicKey b A)"
by blast

lemma analz_symKeys_Decrypt:
     "[| Crypt K X ∈ analz H;  K ∈ symKeys;  Key K ∈ analz H |]
      ==> X ∈ analz H"
by auto
```

3.6 "Image" Equations That Hold for Injective Functions

```
lemma invKey_image_eq [iff]: "(invKey x ∈ invKey`A) = (x∈A)"
by auto
```

holds because invKey is injective

```
lemma publicKey_image_eq [iff]:
     "(publicKey b A ∈ publicKey c ` AS) = (b=c & A∈AS)"
by auto

lemma privateKey_image_eq [iff]:
     "(invKey (publicKey b A) ∈ invKey ` publicKey c ` AS) = (b=c & A∈AS)"
by auto

lemma privateKey_notin_image_publicKey [iff]:
     "invKey (publicKey b A) ∉ publicKey c ` AS"
by auto

lemma publicKey_notin_image_privateKey [iff]:
     "publicKey b A ∉ invKey ` publicKey c ` AS"
by auto

lemma keysFor_parts_initState [simp]: "keysFor (parts (initState C)) = {}"
apply (simp add: keysFor_def)
apply (induct_tac "C")
apply (auto intro: range_eqI)
done
```

for proving `new_keys_not_used`

```
lemma keysFor_parts_insert:
     "[| K ∈ keysFor (parts (insert X H));  X ∈ synth (analz H) |]
      ==> K ∈ keysFor (parts H) | Key (invKey K) ∈ parts H"
by (force dest!:
         parts_insert_subset_Un [THEN keysFor_mono, THEN [2] rev_subsetD]
         analz_subset_parts [THEN keysFor_mono, THEN [2] rev_subsetD]
            intro: analz_into_parts)

lemma Crypt_imp_keysFor [intro]:
     "[|K ∈ symKeys; Crypt K X ∈ H|] ==> K ∈ keysFor H"
by (drule Crypt_imp_invKey_keysFor, simp)
```

Agents see their own private keys!

```
lemma privateKey_in_initStateCA [iff]:
     "Key (invKey (publicKey b A)) ∈ initState A"
by (case_tac "A", auto)
```

Agents see their own public keys!

```
lemma publicKey_in_initStateCA [iff]: "Key (publicKey b A) ∈ initState A"
by (case_tac "A", auto)
```

RCA sees CAs' public keys!

```
lemma pubK_CA_in_initState_RCA [iff]:
     "Key (publicKey b (CA i)) ∈ initState RCA"
by auto
```

Spy knows all public keys

```
lemma knows_Spy_pubEK_i [iff]: "Key (publicKey b A) ∈ knows Spy evs"
```

apply *(induct_tac "evs")*
apply *(simp_all add: imageI knows_Cons split add: event.split)*
done

declare *knows_Spy_pubEK_i [THEN analz.Inj, iff]*

Spy sees private keys of bad agents! [and obviously public keys too]

lemma *knows_Spy_bad_privateKey [intro!]:*
"A ∈ bad ==> Key (invKey (publicKey b A)) ∈ knows Spy evs"
by *(rule initState_subset_knows [THEN subsetD], simp)*

3.7 Fresh Nonces for Possibility Theorems

lemma *Nonce_notin_initState [iff]: "Nonce N ∉ parts (initState B)"*
by *(induct_tac "B", auto)*

lemma *Nonce_notin_used_empty [simp]: "Nonce N ∉ used []"*
by *(simp add: used_Nil)*

In any trace, there is an upper bound N on the greatest nonce in use.

lemma *Nonce_supply_lemma: "∃N. ∀n. N<=n --> Nonce n ∉ used evs"*
apply *(induct_tac "evs")*
apply *(rule_tac x = 0* **in** *exI)*
apply *(simp_all add: used_Cons split add: event.split, safe)*
apply *(rule msg_Nonce_supply [THEN exE], blast elim!: add_leE)+*
done

lemma *Nonce_supply1: "∃N. Nonce N ∉ used evs"*
by *(rule Nonce_supply_lemma [THEN exE], blast)*

lemma *Nonce_supply: "Nonce (@ N. Nonce N ∉ used evs) ∉ used evs"*
apply *(rule Nonce_supply_lemma [THEN exE])*
apply *(rule someI, fast)*
done

3.8 Specialized Methods for Possibility Theorems

ML
```
{*
val Nonce_supply1 = thm "Nonce_supply1";
val Nonce_supply = thm "Nonce_supply";

val used_Says = thm "used_Says";
val used_Notes = thm "used_Notes";

(*Tactic for possibility theorems (Isar interface)*)
fun gen_possibility_tac ss state = state |>
    REPEAT (*omit used_Says so that Nonces start from different traces!*)
    (ALLGOALS (simp_tac (ss delsimps [used_Says,used_Notes]))
     THEN
     REPEAT_FIRST (eq_assume_tac ORELSE'
                   resolve_tac [refl, conjI, Nonce_supply]))
```

```
(*Tactic for possibility theorems (ML script version)*)
fun possibility_tac state = gen_possibility_tac (simpset()) state

(*For harder protocols (such as SET_CR!), where we have to set up some
  nonces and keys initially*)
fun basic_possibility_tac st = st |>
    REPEAT
    (ALLGOALS (asm_simp_tac (simpset() setSolver safe_solver))
     THEN
     REPEAT_FIRST (resolve_tac [refl, conjI]))
*}
```

method_setup *possibility* = {*
    Method.ctxt_args (fn ctxt =>
        Method.METHOD (fn facts =>
            gen_possibility_tac (Simplifier.get_local_simpset ctxt))) *}
    "for proving possibility theorems"

3.9 Specialized Rewriting for Theorems About *analz* and Image

lemma *insert_Key_singleton: "insert (Key K) H = Key ' {K} Un H"*
by *blast*

lemma *insert_Key_image:*
     "insert (Key K) (Key'KK Un C) = Key ' (insert K KK) Un C"
by *blast*

Needed for *DK_fresh_not_KeyCryptKey*

lemma *publicKey_in_used [iff]: "Key (publicKey b A) ∈ used evs"*
by *auto*

lemma *privateKey_in_used [iff]: "Key (invKey (publicKey b A)) ∈ used evs"*
by *(blast intro!: initState_into_used)*

Reverse the normal simplification of "image" to build up (not break down) the set of keys. Based on *analz_image_freshK_ss*, but simpler.

lemmas *analz_image_keys_simps =*
        simp_thms mem_simps — these two allow its use with *only:*
        image_insert [THEN sym] image_Un [THEN sym]
        rangeI symKeys_neq_imp_neq
        insert_Key_singleton insert_Key_image Un_assoc [THEN sym]

3.10 Controlled Unfolding of Abbreviations

A set is expanded only if a relation is applied to it

lemma *def_abbrev_simp_relation:*
```
     "A == B ==> (A ∈ X) = (B ∈ X) &
                 (u = A) = (u = B) &
                 (A = u) = (B = u)"
```
by *auto*

A set is expanded only if one of the given functions is applied to it

```
lemma def_abbrev_simp_function:
     "A == B
      ==> parts (insert A X) = parts (insert B X) &
          analz (insert A X) = analz (insert B X) &
          keysFor (insert A X) = keysFor (insert B X)"
by auto
```

3.10.1 Special Simplification Rules for signCert

Avoids duplicating X and its components!

```
lemma parts_insert_signCert:
     "parts (insert (signCert K X) H) =
      insert {|X, Crypt K X|} (parts (insert (Crypt K X) H))"
by (simp add: signCert_def insert_commute [of X])
```

Avoids a case split! [X is always available]

```
lemma analz_insert_signCert:
     "analz (insert (signCert K X) H) =
      insert {|X, Crypt K X|} (insert (Crypt K X) (analz (insert X H)))"
by (simp add: signCert_def insert_commute [of X])

lemma keysFor_insert_signCert: "keysFor (insert (signCert K X) H) = keysFor
H"
by (simp add: signCert_def)
```

Controlled rewrite rules for `signCert`, just the definitions of the others. Encryption primitives are just expanded, despite their huge redundancy!

```
lemmas abbrev_simps [simp] =
    parts_insert_signCert analz_insert_signCert keysFor_insert_signCert
    sign_def     [THEN def_abbrev_simp_relation]
    sign_def     [THEN def_abbrev_simp_function]
    signCert_def [THEN def_abbrev_simp_relation]
    signCert_def [THEN def_abbrev_simp_function]
    certC_def    [THEN def_abbrev_simp_relation]
    certC_def    [THEN def_abbrev_simp_function]
    cert_def     [THEN def_abbrev_simp_relation]
    cert_def     [THEN def_abbrev_simp_function]
    EXcrypt_def  [THEN def_abbrev_simp_relation]
    EXcrypt_def  [THEN def_abbrev_simp_function]
    EXHcrypt_def [THEN def_abbrev_simp_relation]
    EXHcrypt_def [THEN def_abbrev_simp_function]
    Enc_def      [THEN def_abbrev_simp_relation]
    Enc_def      [THEN def_abbrev_simp_function]
    EncB_def     [THEN def_abbrev_simp_relation]
    EncB_def     [THEN def_abbrev_simp_function]
```

3.10.2 Elimination Rules for Controlled Rewriting

```
lemma Enc_partsE:
     "!!R. [|Enc SK K EK M ∈ parts H;
             [|Crypt K (sign SK M) ∈ parts H;
               Crypt EK (Key K) ∈ parts H|] ==> R|]
           ==> R"
```

```
by (unfold Enc_def, blast)

lemma EncB_partsE:
     "!!R. [|EncB SK K EK M b ∈ parts H;
             [|Crypt K (sign SK {|M, Hash b|}) ∈ parts H;
               Crypt EK (Key K) ∈ parts H;
               b ∈ parts H|] ==> R|]
           ==> R"
by (unfold EncB_def Enc_def, blast)

lemma EXcrypt_partsE:
     "!!R. [|EXcrypt K EK M m ∈ parts H;
             [|Crypt K {|M, Hash m|} ∈ parts H;
               Crypt EK {|Key K, m|} ∈ parts H|] ==> R|]
           ==> R"
by (unfold EXcrypt_def, blast)
```

3.11 Lemmas to Simplify Expressions Involving analz

```
lemma analz_knows_absorb:
     "Key K ∈ analz (knows Spy evs)
      ==> analz (Key ` (insert K H) ∪ knows Spy evs) =
          analz (Key ` H ∪ knows Spy evs)"
by (simp add: analz_insert_eq Un_upper2 [THEN analz_mono, THEN subsetD])

lemma analz_knows_absorb2:
     "Key K ∈ analz (knows Spy evs)
      ==> analz (Key ` (insert X (insert K H)) ∪ knows Spy evs) =
          analz (Key ` (insert X H) ∪ knows Spy evs)"
apply (subst insert_commute)
apply (erule analz_knows_absorb)
done

lemma analz_insert_subset_eq:
     "[|X ∈ analz (knows Spy evs);  knows Spy evs ⊆ H|]
      ==> analz (insert X H) = analz H"
apply (rule analz_insert_eq)
apply (blast intro: analz_mono [THEN [2] rev_subsetD])
done

lemmas analz_insert_simps =
         analz_insert_subset_eq Un_upper2
         subset_insertI [THEN [2] subset_trans]
```

3.12 Freshness Lemmas

```
lemma in_parts_Says_imp_used:
     "[|Key K ∈ parts {X}; Says A B X ∈ set evs|] ==> Key K ∈ used evs"
by (blast intro: parts_trans dest!: Says_imp_knows_Spy [THEN parts.Inj])
```

A useful rewrite rule with analz_image_keys_simps

```
lemma Crypt_notin_image_Key: "Crypt K X ∉ Key ` KK"
by auto
```

```
lemma fresh_notin_analz_knows_Spy:
     "Key K ∉ used evs ==> Key K ∉ analz (knows Spy evs)"
by (auto dest: analz_into_parts)

end
```

4 The SET Cardholder Registration Protocol

```
theory Cardholder_Registration = PublicSET:
```

Note: nonces seem to consist of 20 bytes. That includes both freshness challenges (Chall-EE, etc.) and important secrets (CardSecret, PANsecret)

Simplifications involving `analz_image_keys_simps` appear to have become much slower. The cause is unclear. However, there is a big blow-up and the rewriting is very sensitive to the set of rewrite rules given.

4.1 Predicate Formalizing the Encryption Association between Keys

```
consts
  KeyCryptKey :: "[key, key, event list] => bool"

primrec

KeyCryptKey_Nil:
  "KeyCryptKey DK K [] = False"

KeyCryptKey_Cons:
```

— Says is the only important case. 1st case: CR5, where KC3 encrypts KC2. 2nd case: any use of priEK C. Revision 1.12 has a more complicated version with separate treatment of the dependency of KC1, KC2 and KC3 on priEK (CA i.) Not needed since priEK C is never sent (and so can't be lost except at the start).

```
  "KeyCryptKey DK K (ev # evs) =
   (KeyCryptKey DK K evs |
    (case ev of
      Says A B Z =>
       ((∃N X Y. A ≠ Spy &
                 DK ∈ symKeys &
                 Z = {|Crypt DK {|Agent A, Nonce N, Key K, X|}, Y|}) |
        (∃C. DK = priEK C))
    | Gets A' X => False
    | Notes A' X => False))"
```

4.2 Predicate formalizing the association between keys and nonces

```
consts
  KeyCryptNonce :: "[key, key, event list] => bool"

primrec
```

```
KeyCryptNonce_Nil:
  "KeyCryptNonce EK K [] = False"
```

KeyCryptNonce_Cons:
— Says is the only important case. 1st case: CR3, where KC1 encrypts NC2 (distinct from CR5 due to EXH); 2nd case: CR5, where KC3 encrypts NC3; 3rd case: CR6, where KC2 encrypts NC3; 4th case: CR6, where KC2 encrypts NonceCCA; 5th case: any use of `priEK C` (including CardSecret). NB the only Nonces we need to keep secret are CardSecret and NonceCCA. But we can't prove `Nonce_compromise` unless the relation covers ALL nonces that the protocol keeps secret.

```
  "KeyCryptNonce DK N (ev # evs) =
   (KeyCryptNonce DK N evs |
    (case ev of
      Says A B Z =>
       A ≠ Spy &
       ((∃X Y. DK ∈ symKeys &
               Z = (EXHcrypt DK X {|Agent A, Nonce N|} Y)) |
        (∃X Y. DK ∈ symKeys &
               Z = {|Crypt DK {|Agent A, Nonce N, X|}, Y|}) |
        (∃K i X Y.
          K ∈ symKeys &
          Z = Crypt K {|sign (priSK (CA i)) {|Agent B, Nonce N, X|}, Y|} &
          (DK=K | KeyCryptKey DK K evs)) |
        (∃K C NC3 Y.
          K ∈ symKeys &
          Z = Crypt K
                {|sign (priSK C) {|Agent B, Nonce NC3, Agent C, Nonce N|},
                  Y|} &
          (DK=K | KeyCryptKey DK K evs)) |
        (∃C. DK = priEK C))
    | Gets A' X => False
    | Notes A' X => False))"
```

4.3 Formal protocol definition

consts `set_cr :: "event list set"`
inductive `set_cr`
intros

Nil: — Initial trace is empty
```
"[] ∈ set_cr"
```

Fake: — The spy MAY say anything he CAN say.
```
"[| evsf ∈ set_cr; X ∈ synth (analz (knows Spy evsf)) |]
 ==> Says Spy B X  # evsf ∈ set_cr"
```

Reception: — If A sends a message X to B, then B might receive it
```
"[| evsr ∈ set_cr; Says A B X ∈ set evsr |]
 ==> Gets B X  # evsr ∈ set_cr"
```

SET_CR1: — CardCInitReq: C initiates a run, sending a nonce to CCA
```
"[| evs1 ∈ set_cr;  C = Cardholder k;  Nonce NC1 ∉ used evs1
|]
```

```
             ==> Says C (CA i) {|Agent C, Nonce NC1|} # evs1 ∈ set_cr"
```

`SET_CR2:` — CardCInitRes: CA responds sending NC1 and its certificates

```
            "[| evs2 ∈ set_cr;
                Gets (CA i) {|Agent C, Nonce NC1|} ∈ set evs2 |]
             ==> Says (CA i) C
                      {|sign (priSK (CA i)) {|Agent C, Nonce NC1|},
                        cert (CA i) (pubEK (CA i)) onlyEnc (priSK RCA),
                        cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|}
                   # evs2 ∈ set_cr"
```

`SET_CR3:`

— RegFormReq: C sends his PAN and a new nonce to CA. C verifies that - nonce received is the same as that sent; - certificates are signed by RCA; - certificates are an encryption certificate (flag is onlyEnc) and a signature certificate (flag is onlySig); - certificates pertain to the CA that C contacted (this is done by checking the signature). C generates a fresh symmetric key KC1. The point of encrypting {|`Agent C, Nonce NC2, Hash (Pan (pan C))`|} is not clear.

```
"[| evs3 ∈ set_cr;  C = Cardholder k;
    Nonce NC2 ∉ used evs3;
    Key KC1 ∉ used evs3; KC1 ∈ symKeys;
    Gets C {|sign (invKey SKi) {|Agent X, Nonce NC1|},
             cert (CA i) EKi onlyEnc (priSK RCA),
             cert (CA i) SKi onlySig (priSK RCA)|}
       ∈ set evs3;
    Says C (CA i) {|Agent C, Nonce NC1|} ∈ set evs3|]
 ==> Says C (CA i) (EXHcrypt KC1 EKi {|Agent C, Nonce NC2|} (Pan(pan C)))
       # Notes C {|Key KC1, Agent (CA i)|}
       # evs3 ∈ set_cr"
```

`SET_CR4:`

— RegFormRes: CA responds sending NC2 back with a new nonce NCA, after checking that - the digital envelope is correctly encrypted by `pubEK (CA i)` - the entire message is encrypted with the same key found inside the envelope (here, KC1)

```
"[| evs4 ∈ set_cr;
    Nonce NCA ∉ used evs4;  KC1 ∈ symKeys;
    Gets (CA i) (EXHcrypt KC1 EKi {|Agent C, Nonce NC2|} (Pan(pan X)))
       ∈ set evs4 |]
  ==> Says (CA i) C
          {|sign (priSK (CA i)) {|Agent C, Nonce NC2, Nonce NCA|},
            cert (CA i) (pubEK (CA i)) onlyEnc (priSK RCA),
            cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|}
       # evs4 ∈ set_cr"
```

`SET_CR5:`

— CertReq: C sends his PAN, a new nonce, its proposed public signature key and its half of the secret value to CA. We now assume that C has a fixed key pair, and he submits (pubSK C). The protocol does not require this key to be fresh. The encryption below is actually EncX.

```
"[| evs5 ∈ set_cr;  C = Cardholder k;
    Nonce NC3 ∉ used evs5;  Nonce CardSecret ∉ used evs5; NC3≠CardSecret;
    Key KC2 ∉ used evs5; KC2 ∈ symKeys;
    Key KC3 ∉ used evs5; KC3 ∈ symKeys; KC2≠KC3;
    Gets C {|sign (invKey SKi) {|Agent C, Nonce NC2, Nonce NCA|},
```

```
             cert (CA i) EKi onlyEnc (priSK RCA),
             cert (CA i) SKi onlySig (priSK RCA) |}
        ∈ set evs5;
    Says C (CA i) (EXHcrypt KC1 EKi {|Agent C, Nonce NC2|} (Pan(pan C)))
         ∈ set evs5 |]
==> Says C (CA i)
         {|Crypt KC3
             {|Agent C, Nonce NC3, Key KC2, Key (pubSK C),
               Crypt (priSK C)
                 (Hash {|Agent C, Nonce NC3, Key KC2,
                         Key (pubSK C), Pan (pan C), Nonce CardSecret|})|},
           Crypt EKi {|Key KC3, Pan (pan C), Nonce CardSecret|} |}
    # Notes C {|Key KC2, Agent (CA i)|}
    # Notes C {|Key KC3, Agent (CA i)|}
    # evs5 ∈ set_cr"
```

— CertRes: CA responds sending NC3 back with its half of the secret value, its signature certificate and the new cardholder signature certificate. CA checks to have never certified the key proposed by C. NOTE: In Merchant Registration, the corresponding rule (4) uses the "sign" primitive. The encryption below is actually `EncK`, which is just `Crypt K (sign SK X)`.

```
SET_CR6:
"[| evs6 ∈ set_cr;
    Nonce NonceCCA ∉ used evs6;
    KC2 ∈ symKeys;  KC3 ∈ symKeys;  cardSK ∉ symKeys;
    Notes (CA i) (Key cardSK) ∉ set evs6;
    Gets (CA i)
      {|Crypt KC3 {|Agent C, Nonce NC3, Key KC2, Key cardSK,
                    Crypt (invKey cardSK)
                      (Hash {|Agent C, Nonce NC3, Key KC2,
                              Key cardSK, Pan (pan C), Nonce CardSecret|})|},
        Crypt (pubEK (CA i)) {|Key KC3, Pan (pan C), Nonce CardSecret|} |}
      ∈ set evs6 |]
==> Says (CA i) C
         (Crypt KC2
          {|sign (priSK (CA i))
                 {|Agent C, Nonce NC3, Agent(CA i), Nonce NonceCCA|},
            certC (pan C) cardSK (XOR(CardSecret,NonceCCA)) onlySig (priSK
(CA i)),
            cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|})
      # Notes (CA i) (Key cardSK)
      # evs6 ∈ set_cr"
```

declare *Says_imp_knows_Spy [THEN parts.Inj, dest]*
declare *parts.Body [dest]*
declare *analz_into_parts [dest]*
declare *Fake_parts_insert_in_Un [dest]*

A "possibility property": there are traces that reach the end. An unconstrained proof with many subgoals.

lemma *Says_to_Gets:*

```
     "Says A B X # evs ∈ set_cr ==> Gets B X # Says A B X # evs ∈ set_cr"
by (rule set_cr.Reception, auto)
```

The many nonces and keys generated, some simultaneously, force us to introduce them explicitly as shown below.

```
lemma possibility_CR6:
     "[|NC1 < (NC2::nat);  NC2 < NC3;  NC3 < NCA ;
        NCA < NonceCCA;  NonceCCA < CardSecret;
        KC1 < (KC2::key);  KC2 < KC3;
        KC1 ∈ symKeys;  Key KC1 ∉ used [];
        KC2 ∈ symKeys;  Key KC2 ∉ used [];
        KC3 ∈ symKeys;  Key KC3 ∉ used [];
        C = Cardholder k|]
   ==> ∃evs ∈ set_cr.
       Says (CA i) C
            (Crypt KC2
             {|sign (priSK (CA i))
                    {|Agent C, Nonce NC3, Agent(CA i), Nonce NonceCCA|},
               certC (pan C) (pubSK (Cardholder k)) (XOR(CardSecret,NonceCCA))
                     onlySig (priSK (CA i)),
               cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|})
          ∈ set evs"
apply (intro exI bexI)
apply (rule_tac [2]
       set_cr.Nil
        [THEN set_cr.SET_CR1 [of concl: C i NC1],
         THEN Says_to_Gets,
         THEN set_cr.SET_CR2 [of concl: i C NC1],
         THEN Says_to_Gets,
         THEN set_cr.SET_CR3 [of concl: C i KC1 _ NC2],
         THEN Says_to_Gets,
         THEN set_cr.SET_CR4 [of concl: i C NC2 NCA],
         THEN Says_to_Gets,
         THEN set_cr.SET_CR5 [of concl: C i KC3 NC3 KC2 CardSecret],
         THEN Says_to_Gets,
         THEN set_cr.SET_CR6 [of concl: i C KC2]])
apply (tactic "basic_possibility_tac")
apply (simp_all (no_asm_simp) add: symKeys_neq_imp_neq)
done
```

General facts about message reception

```
lemma Gets_imp_Says:
     "[| Gets B X ∈ set evs; evs ∈ set_cr |] ==> ∃A. Says A B X ∈ set evs"
apply (erule rev_mp)
apply (erule set_cr.induct, auto)
done

lemma Gets_imp_knows_Spy:
     "[| Gets B X ∈ set evs; evs ∈ set_cr |]  ==> X ∈ knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)
declare Gets_imp_knows_Spy [THEN parts.Inj, dest]
```

4.4 Proofs on keys

Spy never sees an agent's private keys! (unless it's bad at start)

```
lemma Spy_see_private_Key [simp]:
     "evs ∈ set_cr
      ==> (Key(invKey (publicKey b A)) ∈ parts(knows Spy evs)) = (A ∈ bad)"
by (erule set_cr.induct, auto)

lemma Spy_analz_private_Key [simp]:
     "evs ∈ set_cr ==>
     (Key(invKey (publicKey b A)) ∈ analz(knows Spy evs)) = (A ∈ bad)"
by auto

declare Spy_see_private_Key [THEN [2] rev_iffD1, dest!]
declare Spy_analz_private_Key [THEN [2] rev_iffD1, dest!]
```

4.5 Begin Piero's Theorems on Certificates

Trivial in the current model, where certificates by RCA are secure

```
lemma Crypt_valid_pubEK:
     "[| Crypt (priSK RCA) {|Agent C, Key EKi, onlyEnc|}
           ∈ parts (knows Spy evs);
         evs ∈ set_cr |] ==> EKi = pubEK C"
apply (erule rev_mp)
apply (erule set_cr.induct, auto)
done

lemma certificate_valid_pubEK:
    "[| cert C EKi onlyEnc (priSK RCA) ∈ parts (knows Spy evs);
        evs ∈ set_cr |]
     ==> EKi = pubEK C"
apply (unfold cert_def signCert_def)
apply (blast dest!: Crypt_valid_pubEK)
done

lemma Crypt_valid_pubSK:
     "[| Crypt (priSK RCA) {|Agent C, Key SKi, onlySig|}
           ∈ parts (knows Spy evs);
         evs ∈ set_cr |] ==> SKi = pubSK C"
apply (erule rev_mp)
apply (erule set_cr.induct, auto)
done

lemma certificate_valid_pubSK:
    "[| cert C SKi onlySig (priSK RCA) ∈ parts (knows Spy evs);
        evs ∈ set_cr |] ==> SKi = pubSK C"
apply (unfold cert_def signCert_def)
apply (blast dest!: Crypt_valid_pubSK)
done

lemma Gets_certificate_valid:
     "[| Gets A {| X, cert C EKi onlyEnc (priSK RCA),
                  cert C SKi onlySig (priSK RCA)|} ∈ set evs;
```

```
        evs ∈ set_cr |]
     ==> EKi = pubEK C & SKi = pubSK C"
by (blast dest: certificate_valid_pubEK certificate_valid_pubSK)
```

Nobody can have used non-existent keys!

```
lemma new_keys_not_used:
     "[|K ∈ symKeys; Key K ∉ used evs; evs ∈ set_cr|]
      ==> K ∉ keysFor (parts (knows Spy evs))"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_cr.induct)
apply (frule_tac [8] Gets_certificate_valid)
apply (frule_tac [6] Gets_certificate_valid, simp_all)
apply (force dest!: usedI keysFor_parts_insert) — Fake
apply (blast,auto)  — Others
done
```

4.6 New versions: as above, but generalized to have the KK argument

```
lemma gen_new_keys_not_used:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_cr |]
      ==> Key K ∉ used evs --> K ∈ symKeys -->
          K ∉ keysFor (parts (Key`KK Un knows Spy evs))"
by (auto simp add: new_keys_not_used)

lemma gen_new_keys_not_analzd:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_cr |]
      ==> K ∉ keysFor (analz (Key`KK Un knows Spy evs))"
by (blast intro: keysFor_mono [THEN [2] rev_subsetD]
          dest: gen_new_keys_not_used)

lemma analz_Key_image_insert_eq:
     "[|K ∈ symKeys; Key K ∉ used evs; evs ∈ set_cr |]
      ==> analz (Key ` (insert K KK) ∪ knows Spy evs) =
          insert (Key K) (analz (Key ` KK ∪ knows Spy evs))"
by (simp add: gen_new_keys_not_analzd)

lemma Crypt_parts_imp_used:
     "[|Crypt K X ∈ parts (knows Spy evs);
        K ∈ symKeys; evs ∈ set_cr |] ==> Key K ∈ used evs"
apply (rule ccontr)
apply (force dest: new_keys_not_used Crypt_imp_invKey_keysFor)
done

lemma Crypt_analz_imp_used:
     "[|Crypt K X ∈ analz (knows Spy evs);
        K ∈ symKeys; evs ∈ set_cr |] ==> Key K ∈ used evs"
by (blast intro: Crypt_parts_imp_used)
```

4.7 Useful lemmas

Rewriting rule for private encryption keys. Analogous rewriting rules for other keys aren't needed.

```
lemma parts_image_priEK:
     "[|Key (priEK C) ∈ parts (Key`KK Un (knows Spy evs));
        evs ∈ set_cr|] ==> priEK C ∈ KK | C ∈ bad"
by auto
```

trivial proof because (priEK C) never appears even in (parts evs)

```
lemma analz_image_priEK:
     "evs ∈ set_cr ==>
          (Key (priEK C) ∈ analz (Key`KK Un (knows Spy evs))) =
          (priEK C ∈ KK | C ∈ bad)"
by (blast dest!: parts_image_priEK intro: analz_mono [THEN [2] rev_subsetD])
```

4.8 Secrecy of Session Keys

4.8.1 Lemmas about the predicate KeyCryptKey

A fresh DK cannot be associated with any other (with respect to a given trace).

```
lemma DK_fresh_not_KeyCryptKey:
     "[| Key DK ∉ used evs; evs ∈ set_cr |] ==> ~ KeyCryptKey DK K evs"
apply (erule rev_mp)
apply (erule set_cr.induct)
apply (simp_all (no_asm_simp))
apply (blast dest: Crypt_analz_imp_used)+
done
```

A fresh K cannot be associated with any other. The assumption that DK isn't a private encryption key may be an artifact of the particular definition of KeyCryptKey.

```
lemma K_fresh_not_KeyCryptKey:
     "[|∀C. DK ≠ priEK C; Key K ∉ used evs|] ==> ~ KeyCryptKey DK K evs"
apply (induct evs)
apply (auto simp add: parts_insert2 split add: event.split)
done
```

This holds because if (priEK (CA i)) appears in any traffic then it must be known to the Spy, by `Spy_see_private_Key`

```
lemma cardSK_neq_priEK:
     "[|Key cardSK ∉ analz (knows Spy evs);
        Key cardSK : parts (knows Spy evs);
        evs ∈ set_cr|] ==> cardSK ≠ priEK C"
by blast

lemma not_KeyCryptKey_cardSK [rule_format (no_asm)]:
     "[|cardSK ∉ symKeys;  ∀C. cardSK ≠ priEK C;  evs ∈ set_cr|] ==>
      Key cardSK ∉ analz (knows Spy evs) --> ~ KeyCryptKey cardSK K evs"
by (erule set_cr.induct, analz_mono_contra, auto)
```

Lemma for message 5: pubSK C is never used to encrypt Keys.

```
lemma pubSK_not_KeyCryptKey [simp]: "~ KeyCryptKey (pubSK C) K evs"
apply (induct_tac "evs")
apply (auto simp add: parts_insert2 split add: event.split)
done
```

Lemma for message 6: either cardSK is compromised (when we don't care) or else cardSK hasn't been used to encrypt K. Previously we treated message 5 in the same way, but the current model assumes that rule `SET_CR5` is executed only by honest agents.

```
lemma msg6_KeyCryptKey_disj:
     "[|Gets B {|Crypt KC3 {|Agent C, Nonce N, Key KC2, Key cardSK, X|}, Y|}
          ∈ set evs;
        cardSK ∉ symKeys;  evs ∈ set_cr|]
      ==> Key cardSK ∈ analz (knows Spy evs) |
          (∀K. ~ KeyCryptKey cardSK K evs)"
by (blast dest: not_KeyCryptKey_cardSK intro: cardSK_neq_priEK)
```

As usual: we express the property as a logical equivalence

```
lemma Key_analz_image_Key_lemma:
     "P --> (Key K ∈ analz (Key`KK Un H)) --> (K ∈ KK | Key K ∈ analz H)
      ==>
      P --> (Key K ∈ analz (Key`KK Un H)) = (K ∈ KK | Key K ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])
```

```
ML
{*
val Gets_certificate_valid = thm "Gets_certificate_valid";

fun valid_certificate_tac i =
    EVERY [ftac Gets_certificate_valid i,
           assume_tac i,
           etac conjE i, REPEAT (hyp_subst_tac i)];
*}
```

The `(no_asm)` attribute is essential, since it retains the quantifier and allows the simprule's condition to itself be simplified.

```
lemma symKey_compromise [rule_format (no_asm)]:
     "evs ∈ set_cr ==>
      (∀SK KK. SK ∈ symKeys ⟶ (∀K ∈ KK. ~ KeyCryptKey K SK evs)   -->
               (Key SK ∈ analz (Key`KK Un (knows Spy evs))) =
               (SK ∈ KK | Key SK ∈ analz (knows Spy evs)))"
apply (erule set_cr.induct)
apply (rule_tac [!] allI) +
apply (rule_tac [!] impI [THEN Key_analz_image_Key_lemma, THEN impI])+
apply (tactic{*valid_certificate_tac 8*}) — for message 5
apply (tactic{*valid_certificate_tac 6*}) — for message 5
apply (erule_tac [9] msg6_KeyCryptKey_disj [THEN disjE])
apply (simp_all
         del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps analz_knows_absorb
              analz_Key_image_insert_eq notin_image_iff
              K_fresh_not_KeyCryptKey
              DK_fresh_not_KeyCryptKey ball_conj_distrib
```

```
              analz_image_priEK disj_simps)
```
— 46 seconds on a 1.8GHz machine

```
apply spy_analz
apply blast  — 3
apply blast  — 5
done
```

The remaining quantifiers seem to be essential. NO NEED to assume the cardholder's OK: bad cardholders don't do anything wrong!!

```
lemma symKey_secrecy [rule_format]:
     "[|CA i ∉ bad;  K ∈ symKeys;  evs ∈ set_cr|]
      ==> ∀X c. Says (Cardholder c) (CA i) X ∈ set evs -->
                Key K ∈ parts{X} -->
                Cardholder c ∉ bad -->
                Key K ∉ analz (knows Spy evs)"
apply (erule set_cr.induct)
apply (frule_tac [8] Gets_certificate_valid) — for message 5
apply (frule_tac [6] Gets_certificate_valid) — for message 3
apply (erule_tac [11] msg6_KeyCryptKey_disj [THEN disjE])
apply (simp_all del: image_insert image_Un imp_disjL
         add: symKey_compromise fresh_notin_analz_knows_Spy
              analz_image_keys_simps analz_knows_absorb
              analz_Key_image_insert_eq notin_image_iff
              K_fresh_not_KeyCryptKey
              DK_fresh_not_KeyCryptKey
              analz_image_priEK)
```
— 13 seconds on a 1.8GHz machine

```
apply spy_analz  — Fake
apply (auto intro: analz_into_parts [THEN usedI] in_parts_Says_imp_used)
done
```

4.9 Primary Goals of Cardholder Registration

The cardholder's certificate really was created by the CA, provided the CA is uncompromised

Lemma concerning the actual signed message digest

```
lemma cert_valid_lemma:
     "[|Crypt (priSK (CA i)) {|Hash {|Nonce N, Pan(pan C)|}, Key cardSK, N1|}
          ∈ parts (knows Spy evs);
        CA i ∉ bad; evs ∈ set_cr|]
  ==> ∃KC2 X Y. Says (CA i) C
                     (Crypt KC2
                       {|X, certC (pan C) cardSK N onlySig (priSK (CA i)),
Y|})
                  ∈ set evs"
apply (erule rev_mp)
apply (erule set_cr.induct)
apply (simp_all (no_asm_simp))
apply auto
done
```

Pre-packaged version for cardholder. We don't try to confirm the values of KC2, X and Y, since they are not important.

```
lemma certificate_valid_cardSK:
    "[|Gets C (Crypt KC2 {|X, certC (pan C) cardSK N onlySig (invKey SKi),
                              cert (CA i) SKi onlySig (priSK RCA)|}) ∈ set
evs;
        CA i ∉ bad; evs ∈ set_cr|]
  ==> ∃KC2 X Y. Says (CA i) C
                     (Crypt KC2
                       {|X, certC (pan C) cardSK N onlySig (priSK (CA i)),
Y|})
                  ∈ set evs"
by (force dest!: Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Body]
                    certificate_valid_pubSK cert_valid_lemma)
```

```
lemma Hash_imp_parts [rule_format]:
     "evs ∈ set_cr
      ==> Hash{|X, Nonce N|} ∈ parts (knows Spy evs) -->
          Nonce N ∈ parts (knows Spy evs)"
apply (erule set_cr.induct, force)
apply (simp_all (no_asm_simp))
apply (blast intro: parts_mono [THEN [2] rev_subsetD])
done
```

```
lemma Hash_imp_parts2 [rule_format]:
     "evs ∈ set_cr
      ==> Hash{|X, Nonce M, Y, Nonce N|} ∈ parts (knows Spy evs) -->
          Nonce M ∈ parts (knows Spy evs) & Nonce N ∈ parts (knows Spy evs)"
apply (erule set_cr.induct, force)
apply (simp_all (no_asm_simp))
apply (blast intro: parts_mono [THEN [2] rev_subsetD])
done
```

4.10 Secrecy of Nonces

4.10.1 Lemmas about the predicate KeyCryptNonce

A fresh DK cannot be associated with any other (with respect to a given trace).

```
lemma DK_fresh_not_KeyCryptNonce:
     "[| DK ∈ symKeys; Key DK ∉ used evs; evs ∈ set_cr |]
      ==> ~ KeyCryptNonce DK K evs"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_cr.induct)
apply (simp_all (no_asm_simp))
apply blast
apply blast
apply (auto simp add: DK_fresh_not_KeyCryptKey)
done
```

A fresh N cannot be associated with any other (with respect to a given trace).

```
lemma N_fresh_not_KeyCryptNonce:
     "∀C. DK ≠ priEK C ==> Nonce N ∉ used evs --> ~ KeyCryptNonce DK N evs"
apply (induct_tac "evs")
```

```
apply (case_tac [2] "a")
apply (auto simp add: parts_insert2)
done

lemma not_KeyCryptNonce_cardSK [rule_format (no_asm)]:
     "[|cardSK ∉ symKeys;  ∀C. cardSK ≠ priEK C;  evs ∈ set_cr|] ==>
      Key cardSK ∉ analz (knows Spy evs) --> ~ KeyCryptNonce cardSK N evs"
apply (erule set_cr.induct, analz_mono_contra, simp_all)
apply (blast dest: not_KeyCryptKey_cardSK)  — 6
done
```

4.10.2 Lemmas for message 5 and 6: either cardSK is compromised (when we don't care) or else cardSK hasn't been used to encrypt K.

Lemma for message 5: pubSK C is never used to encrypt Nonces.

```
lemma pubSK_not_KeyCryptNonce [simp]: "~ KeyCryptNonce (pubSK C) N evs"
apply (induct_tac "evs")
apply (auto simp add: parts_insert2 split add: event.split)
done
```

Lemma for message 6: either cardSK is compromised (when we don't care) or else cardSK hasn't been used to encrypt K.

```
lemma msg6_KeyCryptNonce_disj:
     "[|Gets B {|Crypt KC3 {|Agent C, Nonce N, Key KC2, Key cardSK, X|}, Y|}
          ∈ set evs;
        cardSK ∉ symKeys;  evs ∈ set_cr|]
      ==> Key cardSK ∈ analz (knows Spy evs) |
          ((∀K. ~ KeyCryptKey cardSK K evs) &
           (∀N. ~ KeyCryptNonce cardSK N evs))"
by (blast dest: not_KeyCryptKey_cardSK not_KeyCryptNonce_cardSK
          intro: cardSK_neq_priEK)
```

As usual: we express the property as a logical equivalence

```
lemma Nonce_analz_image_Key_lemma:
     "P --> (Nonce N ∈ analz (Key`KK Un H)) --> (Nonce N ∈ analz H)
      ==> P --> (Nonce N ∈ analz (Key`KK Un H)) = (Nonce N ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])
```

The (no_asm) attribute is essential, since it retains the quantifier and allows the simprule's condition to itself be simplified.

```
lemma Nonce_compromise [rule_format (no_asm)]:
     "evs ∈ set_cr ==>
      (∀N KK. (∀K ∈ KK. ~ KeyCryptNonce K N evs)   -->
              (Nonce N ∈ analz (Key`KK Un (knows Spy evs))) =
              (Nonce N ∈ analz (knows Spy evs)))"
apply (erule set_cr.induct)
apply (rule_tac [!] allI)+
apply (rule_tac [!] impI [THEN Nonce_analz_image_Key_lemma])+
apply (frule_tac [8] Gets_certificate_valid) — for message 5
apply (frule_tac [6] Gets_certificate_valid) — for message 3
apply (frule_tac [11] msg6_KeyCryptNonce_disj)
```

```
apply (erule_tac [13] disjE)
apply (simp_all del: image_insert image_Un
         add: symKey_compromise
              analz_image_keys_simps analz_knows_absorb
              analz_Key_image_insert_eq notin_image_iff
              N_fresh_not_KeyCryptNonce
              DK_fresh_not_KeyCryptNonce K_fresh_not_KeyCryptKey
              ball_conj_distrib analz_image_priEK)
  — 71 seconds on a 1.8GHz machine
apply spy_analz  — Fake
apply blast  — 3
apply blast  — 5
```

Message 6

```
apply (force del: allE ballE impCE simp add: symKey_compromise)
  — cardSK compromised
```

Simplify again–necessary because the previous simplification introduces some logical connectives

```
apply (force del: allE ballE impCE
          simp del: image_insert image_Un imp_disjL
          simp add: analz_image_keys_simps symKey_compromise)
done
```

4.11 Secrecy of CardSecret: the Cardholder's secret

```
lemma NC2_not_CardSecret:
     "[|Crypt EKj {|Key K, Pan p, Hash {|Agent D, Nonce N|}|}
          ∈ parts (knows Spy evs);
        Key K ∉ analz (knows Spy evs);
        Nonce N ∉ analz (knows Spy evs);
       evs ∈ set_cr|]
      ==> Crypt EKi {|Key K', Pan p', Nonce N|} ∉ parts (knows Spy evs)"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_cr.induct, analz_mono_contra, simp_all)
apply (blast dest: Hash_imp_parts)+
done

lemma KC2_secure_lemma [rule_format]:
     "[|U = Crypt KC3 {|Agent C, Nonce N, Key KC2, X|};
        U ∈ parts (knows Spy evs);
        evs ∈ set_cr|]
  ==> Nonce N ∉ analz (knows Spy evs) -->
      (∃k i W. Says (Cardholder k) (CA i) {|U,W|} ∈ set evs &
               Cardholder k ∉ bad & CA i ∉ bad)"
apply (erule_tac P = "U ∈ ?H" in rev_mp)
apply (erule set_cr.induct)
apply (tactic{*valid_certificate_tac 8*})  — for message 5
apply (simp_all del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps analz_knows_absorb
              analz_knows_absorb2 notin_image_iff)
  — 19 seconds on a 1.8GHz machine
```

```
apply (simp_all (no_asm_simp)) — leaves 4 subgoals
apply (blast intro!: analz_insertI)+
done

lemma KC2_secrecy:
     "[|Gets B {|Crypt K {|Agent C, Nonce N, Key KC2, X|}, Y|} ∈ set evs;
        Nonce N ∉ analz (knows Spy evs);  KC2 ∈ symKeys;
        evs ∈ set_cr|]
       ==> Key KC2 ∉ analz (knows Spy evs)"
by (force dest!: refl [THEN KC2_secure_lemma] symKey_secrecy)
```

Inductive version

```
lemma CardSecret_secrecy_lemma [rule_format]:
     "[|CA i ∉ bad;  evs ∈ set_cr|]
      ==> Key K ∉ analz (knows Spy evs) -->
          Crypt (pubEK (CA i)) {|Key K, Pan p, Nonce CardSecret|}
             ∈ parts (knows Spy evs) -->
          Nonce CardSecret ∉ analz (knows Spy evs)"
apply (erule set_cr.induct, analz_mono_contra)
apply (tactic{*valid_certificate_tac 8*}) — for message 5
apply (tactic{*valid_certificate_tac 6*}) — for message 5
apply (frule_tac [9] msg6_KeyCryptNonce_disj [THEN disjE])
apply (simp_all
         del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps analz_knows_absorb
              analz_Key_image_insert_eq notin_image_iff
              EXHcrypt_def Crypt_notin_image_Key
              N_fresh_not_KeyCryptNonce DK_fresh_not_KeyCryptNonce
              ball_conj_distrib Nonce_compromise symKey_compromise
              analz_image_priEK)
  — 12 seconds on a 1.8GHz machine
apply spy_analz  — Fake
apply (simp_all (no_asm_simp))
apply blast  — 1
apply (blast dest!: Gets_imp_knows_Spy [THEN analz.Inj])  — 2
apply blast  — 3
apply (blast dest: NC2_not_CardSecret Gets_imp_knows_Spy [THEN analz.Inj]
analz_symKeys_Decrypt)  — 4
apply blast  — 5
apply (blast dest: KC2_secrecy)+  — Message 6: two cases
done
```

Packaged version for cardholder

```
lemma CardSecret_secrecy:
     "[|Cardholder k ∉ bad;  CA i ∉ bad;
        Says (Cardholder k) (CA i)
           {|X, Crypt EKi {|Key KC3, Pan p, Nonce CardSecret|}|} ∈ set evs;
        Gets A {|Z, cert (CA i) EKi onlyEnc (priSK RCA),
                    cert (CA i) SKi onlySig (priSK RCA)|} ∈ set evs;
        KC3 ∈ symKeys;  evs ∈ set_cr|]
      ==> Nonce CardSecret ∉ analz (knows Spy evs)"
apply (frule Gets_certificate_valid, assumption)
apply (subgoal_tac "Key KC3 ∉ analz (knows Spy evs) ")
apply (blast dest: CardSecret_secrecy_lemma)
```

```
apply (rule symKey_secrecy)
apply (auto simp add: parts_insert2)
done
```

4.12 Secrecy of NonceCCA [the CA's secret]

```
lemma NC2_not_NonceCCA:
     "[|Hash {|Agent C', Nonce N', Agent C, Nonce N|}
          ∈ parts (knows Spy evs);
        Nonce N ∉ analz (knows Spy evs);
       evs ∈ set_cr|]
      ==> Crypt KC1 {|{|Agent B, Nonce N|}, Hash p|} ∉ parts (knows Spy evs)"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_cr.induct, analz_mono_contra, simp_all)
apply (blast dest: Hash_imp_parts2)+
done
```

Inductive version

```
lemma NonceCCA_secrecy_lemma [rule_format]:
     "[|CA i ∉ bad;  evs ∈ set_cr|]
      ==> Key K ∉ analz (knows Spy evs) -->
          Crypt K
            {|sign (priSK (CA i))
                   {|Agent C, Nonce N, Agent(CA i), Nonce NonceCCA|},
              X, Y|}
            ∈ parts (knows Spy evs) -->
          Nonce NonceCCA ∉ analz (knows Spy evs)"
apply (erule set_cr.induct, analz_mono_contra)
apply (tactic{*valid_certificate_tac 8*}) — for message 5
apply (tactic{*valid_certificate_tac 6*}) — for message 5
apply (frule_tac [9] msg6_KeyCryptNonce_disj [THEN disjE])
apply (simp_all
         del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps analz_knows_absorb sign_def
              analz_Key_image_insert_eq notin_image_iff
              EXHcrypt_def Crypt_notin_image_Key
              N_fresh_not_KeyCryptNonce DK_fresh_not_KeyCryptNonce
              ball_conj_distrib Nonce_compromise symKey_compromise
              analz_image_priEK)
  — 15 seconds on a 1.8GHz machine
apply spy_analz  — Fake
apply blast  — 1
apply (blast dest!: Gets_imp_knows_Spy [THEN analz.Inj])  — 2
apply blast  — 3
apply (blast dest: NC2_not_NonceCCA)  — 4
apply blast  — 5
apply (blast dest: KC2_secrecy)+  — Message 6: two cases
done
```

Packaged version for cardholder

```
lemma NonceCCA_secrecy:
     "[|Cardholder k ∉ bad;  CA i ∉ bad;
```

```
        Gets (Cardholder k)
           (Crypt KC2
            {|sign (priSK (CA i)) {|Agent C, Nonce N, Agent(CA i), Nonce NonceCCA|},
              X, Y|}) ∈ set evs;
        Says (Cardholder k) (CA i)
           {|Crypt KC3 {|Agent C, Nonce NC3, Key KC2, X'|}, Y'|} ∈ set evs;
        Gets A {|Z, cert (CA i) EKi onlyEnc (priSK RCA),
                    cert (CA i) SKi onlySig (priSK RCA)|} ∈ set evs;
        KC2 ∈ symKeys;  evs ∈ set_cr|]
      ==> Nonce NonceCCA ∉ analz (knows Spy evs)"
apply (frule Gets_certificate_valid, assumption)
apply (subgoal_tac "Key KC2 ∉ analz (knows Spy evs) ")
apply (blast dest: NonceCCA_secrecy_lemma)
apply (rule symKey_secrecy)
apply (auto simp add: parts_insert2)
done
```

We don't bother to prove guarantees for the CA. He doesn't care about the PANSecret: it isn't his credit card!

4.13 Rewriting Rule for PANs

Lemma for message 6: either cardSK isn't a CA's private encryption key, or if it is then (because it appears in traffic) that CA is bad, and so the Spy knows that key already. Either way, we can simplify the expression *analz (insert (Key cardSK) X)*.

```
lemma msg6_cardSK_disj:
     "[|Gets A {|Crypt K {|c, n, k', Key cardSK, X|}, Y|}
          ∈ set evs;  evs ∈ set_cr |]
      ==> cardSK ∉ range(invKey o pubEK o CA) | Key cardSK ∈ knows Spy evs"
by auto

lemma analz_image_pan_lemma:
     "(Pan P ∈ analz (Key`nE Un H)) --> (Pan P ∈ analz H)  ==>
      (Pan P ∈ analz (Key`nE Un H)) =   (Pan P ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])

lemma analz_image_pan [rule_format]:
     "evs ∈ set_cr ==>
       ∀KK. KK <= - invKey ` pubEK ` range CA -->
            (Pan P ∈ analz (Key`KK Un (knows Spy evs))) =
            (Pan P ∈ analz (knows Spy evs))"
apply (erule set_cr.induct)
apply (rule_tac [!] allI impI)+
apply (rule_tac [!] analz_image_pan_lemma)
apply (tactic{*valid_certificate_tac 8*}) — for message 5
apply (tactic{*valid_certificate_tac 6*}) — for message 5
apply (erule_tac [9] msg6_cardSK_disj [THEN disjE])
apply (simp_all
         del: image_insert image_Un
         add: analz_image_keys_simps disjoint_image_iff
              notin_image_iff analz_image_priEK)
  — 33 seconds on a 1.8GHz machine
```

apply *spy_analz*
apply *(simp add: insert_absorb)* — 6
done

```
lemma analz_insert_pan:
     "[| evs ∈ set_cr;  K ∉ invKey ‘ pubEK ‘ range CA |] ==>
          (Pan P ∈ analz (insert (Key K) (knows Spy evs))) =
          (Pan P ∈ analz (knows Spy evs))"
by (simp del: image_insert image_Un
         add: analz_image_keys_simps analz_image_pan)
```

Confidentiality of the PAN. Maybe we could combine the statements of this theorem with *analz_image_pan*, requiring a single induction but a much more difficult proof.

```
lemma pan_confidentiality:
     "[| Pan (pan C) ∈ analz(knows Spy evs); C ≠Spy; evs :set_cr|]
    ==> ∃i X K HN.
        Says C (CA i) {|X, Crypt (pubEK (CA i)) {|Key K, Pan (pan C), HN|}
|}
           ∈ set evs
      & (CA i) ∈ bad"
apply (erule rev_mp)
apply (erule set_cr.induct)
apply (tactic{*valid_certificate_tac 8*}) — for message 5
apply (tactic{*valid_certificate_tac 6*}) — for message 5
apply (erule_tac [9] msg6_cardSK_disj [THEN disjE])
apply (simp_all
         del: image_insert image_Un
         add: analz_image_keys_simps analz_insert_pan analz_image_pan
              notin_image_iff analz_image_priEK)
  — 18 seconds on a 1.8GHz machine
apply spy_analz  — fake
apply blast  — 3
apply blast  — 5
apply (simp (no_asm_simp) add: insert_absorb)  — 6
done
```

4.14 Unicity

```
lemma CR6_Says_imp_Notes:
     "[|Says (CA i) C (Crypt KC2
          {|sign (priSK (CA i)) {|Agent C, Nonce NC3, Agent (CA i), Nonce Y|},
            certC (pan C) cardSK X onlySig (priSK (CA i)),
            cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|})  ∈ set evs;
        evs ∈ set_cr |]
      ==> Notes (CA i) (Key cardSK) ∈ set evs"
apply (erule rev_mp)
apply (erule set_cr.induct)
apply (simp_all (no_asm_simp))
done
```

Unicity of cardSK: it uniquely identifies the other components. This holds because a CA accepts a cardSK at most once.

lemma *cardholder_key_unicity:*
```
     "[|Says (CA i) C (Crypt KC2
          {|sign (priSK (CA i)) {|Agent C, Nonce NC3, Agent (CA i), Nonce Y|},
            certC (pan C) cardSK X onlySig (priSK (CA i)),
            cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|})
          ∈ set evs;
        Says (CA i) C' (Crypt KC2'
          {|sign (priSK (CA i)) {|Agent C', Nonce NC3', Agent (CA i), Nonce
Y'|},
            certC (pan C') cardSK X' onlySig (priSK (CA i)),
            cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|})
          ∈ set evs;
        evs ∈ set_cr |] ==> C=C' & NC3=NC3' & X=X' & KC2=KC2' & Y=Y'"
```
apply *(erule rev_mp)*
apply *(erule rev_mp)*
apply *(erule set_cr.induct)*
apply *(simp_all (no_asm_simp))*
apply *(blast dest!: CR6_Says_imp_Notes)*
done

Cannot show cardSK to be secret because it isn't assumed to be fresh it could be a previously compromised cardSK [e.g. involving a bad CA]

end

5 The SET Merchant Registration Protocol

theory *Merchant_Registration = PublicSET:*

Copmpared with Cardholder Reigstration, *KeyCryptKey* is not needed: no session key encrypts another. Instead we prove the "key compromise" theorems for sets KK that contain no private encryption keys (*priEK C*).

consts *set_mr :: "event list set"*
inductive *set_mr*
intros

Nil: — Initial trace is empty
```
"[] ∈ set_mr"
```

Fake: — The spy MAY say anything he CAN say.
```
"[| evsf ∈ set_mr; X ∈ synth (analz (knows Spy evsf)) |]
 ==> Says Spy B X  # evsf ∈ set_mr"
```

Reception: — If A sends a message X to B, then B might receive it
```
"[| evsr ∈ set_mr; Says A B X ∈ set evsr |]
 ==> Gets B X  # evsr ∈ set_mr"
```

`SET_MR1:` — RegFormReq: M requires a registration form to a CA

```
"[| evs1 ∈ set_mr; M = Merchant k; Nonce NM1 ∉ used evs1 |]
 ==> Says M (CA i) {|Agent M, Nonce NM1|} # evs1 ∈ set_mr"
```

`SET_MR2:` — RegFormRes: CA replies with the registration form and the certificates for her keys

```
"[| evs2 ∈ set_mr; Nonce NCA ∉ used evs2;
    Gets (CA i) {|Agent M, Nonce NM1|} ∈ set evs2 |]
 ==> Says (CA i) M {|sign (priSK (CA i)) {|Agent M, Nonce NM1, Nonce NCA|},
                    cert (CA i) (pubEK (CA i)) onlyEnc (priSK RCA),
                    cert (CA i) (pubSK (CA i)) onlySig (priSK RCA) |}
       # evs2 ∈ set_mr"
```

`SET_MR3:`

— CertReq: M submits the key pair to be certified. The Notes event allows KM1 to be lost if M is compromised. Piero remarks that the agent mentioned inside the signature is not verified to correspond to M. As in CR, each Merchant has fixed key pairs. M is only optionally required to send NCA back, so M doesn't do so in the model

```
"[| evs3 ∈ set_mr; M = Merchant k; Nonce NM2 ∉ used evs3;
    Key KM1 ∉ used evs3;  KM1 ∈ symKeys;
    Gets M {|sign (invKey SKi) {|Agent X, Nonce NM1, Nonce NCA|},
             cert (CA i) EKi onlyEnc (priSK RCA),
             cert (CA i) SKi onlySig (priSK RCA) |}
      ∈ set evs3;
    Says M (CA i) {|Agent M, Nonce NM1|} ∈ set evs3 |]
 ==> Says M (CA i)
         {|Crypt KM1 (sign (priSK M) {|Agent M, Nonce NM2,
                                      Key (pubSK M), Key (pubEK M)|}),
           Crypt EKi (Key KM1)|}
       # Notes M {|Key KM1, Agent (CA i)|}
       # evs3 ∈ set_mr"
```

`SET_MR4:`

— CertRes: CA issues the certificates for merSK and merEK, while checking never to have certified the m even separately. NOTE: In Cardholder Registration the corresponding rule (6) doesn't use the "sign" primitive. "The CertRes shall be signed but not encrypted if the EE is a Merchant or Payment Gateway."– Programmer's Guide, page 191.

```
"[| evs4 ∈ set_mr; M = Merchant k;
    merSK ∉ symKeys;  merEK ∉ symKeys;
    Notes (CA i) (Key merSK) ∉ set evs4;
    Notes (CA i) (Key merEK) ∉ set evs4;
    Gets (CA i) {|Crypt KM1 (sign (invKey merSK)
                         {|Agent M, Nonce NM2, Key merSK, Key merEK|}),
                  Crypt (pubEK (CA i)) (Key KM1) |}
      ∈ set evs4 |]
 ==> Says (CA i) M {|sign (priSK(CA i)) {|Agent M, Nonce NM2, Agent(CA i)|},
                    cert  M      merSK    onlySig (priSK (CA i)),
                    cert  M      merEK    onlyEnc (priSK (CA i)),
                    cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|}
       # Notes (CA i) (Key merSK)
       # Notes (CA i) (Key merEK)
```

```
        # evs4 ∈ set_mr"
```

Note possibility proofs are missing.

```
declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]
```

General facts about message reception

```
lemma Gets_imp_Says:
     "[| Gets B X ∈ set evs; evs ∈ set_mr |] ==> ∃A. Says A B X ∈ set evs"
apply (erule rev_mp)
apply (erule set_mr.induct, auto)
done

lemma Gets_imp_knows_Spy:
     "[| Gets B X ∈ set evs; evs ∈ set_mr |]  ==> X ∈ knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)


declare Gets_imp_knows_Spy [THEN parts.Inj, dest]
```

5.0.1 Proofs on keys

Spy never sees an agent's private keys! (unless it's bad at start)

```
lemma Spy_see_private_Key [simp]:
     "evs ∈ set_mr
      ==> (Key(invKey (publicKey b A)) ∈ parts(knows Spy evs)) = (A ∈ bad)"
apply (erule set_mr.induct)
apply (auto dest!: Gets_imp_knows_Spy [THEN parts.Inj])
done

lemma Spy_analz_private_Key [simp]:
     "evs ∈ set_mr ==>
     (Key(invKey (publicKey b A)) ∈ analz(knows Spy evs)) = (A ∈ bad)"
by auto

declare Spy_see_private_Key [THEN [2] rev_iffD1, dest!]
declare Spy_analz_private_Key [THEN [2] rev_iffD1, dest!]
```

Proofs on certificates - they hold, as in CR, because RCA's keys are secure

```
lemma Crypt_valid_pubEK:
     "[| Crypt (priSK RCA) {|Agent (CA i), Key EKi, onlyEnc|}
           ∈ parts (knows Spy evs);
         evs ∈ set_mr |] ==> EKi = pubEK (CA i)"
apply (erule rev_mp)
apply (erule set_mr.induct, auto)
done

lemma certificate_valid_pubEK:
    "[| cert (CA i) EKi onlyEnc (priSK RCA) ∈ parts (knows Spy evs);
        evs ∈ set_mr |]
```

```
     ==> EKi = pubEK (CA i)"
apply (unfold cert_def signCert_def)
apply (blast dest!: Crypt_valid_pubEK)
done

lemma Crypt_valid_pubSK:
     "[| Crypt (priSK RCA) {|Agent (CA i), Key SKi, onlySig|}
           ∈ parts (knows Spy evs);
         evs ∈ set_mr |] ==> SKi = pubSK (CA i)"
apply (erule rev_mp)
apply (erule set_mr.induct, auto)
done

lemma certificate_valid_pubSK:
    "[| cert (CA i) SKi onlySig (priSK RCA) ∈ parts (knows Spy evs);
        evs ∈ set_mr |] ==> SKi = pubSK (CA i)"
apply (unfold cert_def signCert_def)
apply (blast dest!: Crypt_valid_pubSK)
done

lemma Gets_certificate_valid:
     "[| Gets A {| X, cert (CA i) EKi onlyEnc (priSK RCA),
                      cert (CA i) SKi onlySig (priSK RCA)|} ∈ set evs;
         evs ∈ set_mr |]
      ==> EKi = pubEK (CA i) & SKi = pubSK (CA i)"
by (blast dest: certificate_valid_pubEK certificate_valid_pubSK)
```

Nobody can have used non-existent keys!

```
lemma new_keys_not_used [rule_format,simp]:
     "evs ∈ set_mr
      ==> Key K ∉ used evs --> K ∈ symKeys -->
          K ∉ keysFor (parts (knows Spy evs))"
apply (erule set_mr.induct, simp_all)
apply (force dest!: usedI keysFor_parts_insert)  — Fake
apply force  — Message 2
apply (blast dest: Gets_certificate_valid)  — Message 3
apply force  — Message 4
done
```

5.0.2 New Versions: As Above, but Generalized with the Kk Argument

```
lemma gen_new_keys_not_used [rule_format]:
     "evs ∈ set_mr
      ==> Key K ∉ used evs --> K ∈ symKeys -->
          K ∉ keysFor (parts (Key`KK Un knows Spy evs))"
by auto

lemma gen_new_keys_not_analzd:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_mr |]
      ==> K ∉ keysFor (analz (Key`KK Un knows Spy evs))"
by (blast intro: keysFor_mono [THEN [2] rev_subsetD]
          dest: gen_new_keys_not_used)
```

```
lemma analz_Key_image_insert_eq:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_mr |]
      ==> analz (Key ` (insert K KK) ∪ knows Spy evs) =
          insert (Key K) (analz (Key ` KK ∪ knows Spy evs))"
by (simp add: gen_new_keys_not_analzd)


lemma Crypt_parts_imp_used:
     "[|Crypt K X ∈ parts (knows Spy evs);
        K ∈ symKeys; evs ∈ set_mr |] ==> Key K ∈ used evs"
apply (rule ccontr)
apply (force dest: new_keys_not_used Crypt_imp_invKey_keysFor)
done

lemma Crypt_analz_imp_used:
     "[|Crypt K X ∈ analz (knows Spy evs);
        K ∈ symKeys; evs ∈ set_mr |] ==> Key K ∈ used evs"
by (blast intro: Crypt_parts_imp_used)
```

Rewriting rule for private encryption keys. Analogous rewriting rules for other keys aren't needed.

```
lemma parts_image_priEK:
     "[|Key (priEK (CA i)) ∈ parts (Key`KK Un (knows Spy evs));
        evs ∈ set_mr|] ==> priEK (CA i) ∈ KK | CA i ∈ bad"
by auto
```

trivial proof because (priEK (CA i)) never appears even in (parts evs)

```
lemma analz_image_priEK:
     "evs ∈ set_mr ==>
          (Key (priEK (CA i)) ∈ analz (Key`KK Un (knows Spy evs))) =
          (priEK (CA i) ∈ KK | CA i ∈ bad)"
by (blast dest!: parts_image_priEK intro: analz_mono [THEN [2] rev_subsetD])
```

5.1 Secrecy of Session Keys

This holds because if (priEK (CA i)) appears in any traffic then it must be known to the Spy, by `Spy_see_private_Key`

```
lemma merK_neq_priEK:
     "[|Key merK ∉ analz (knows Spy evs);
        Key merK ∈ parts (knows Spy evs);
        evs ∈ set_mr|] ==> merK ≠ priEK C"
by blast
```

Lemma for message 4: either merK is compromised (when we don't care) or else merK hasn't been used to encrypt K.

```
lemma msg4_priEK_disj:
     "[|Gets B {|Crypt KM1
                   (sign K {|Agent M, Nonce NM2, Key merSK, Key merEK|}),
               Y|} ∈ set evs;
        evs ∈ set_mr|]
  ==> (Key merSK ∈ analz (knows Spy evs) | merSK ∉ range(λC. priEK C))
   &  (Key merEK ∈ analz (knows Spy evs) | merEK ∉ range(λC. priEK C))"
```

```
apply (unfold sign_def)
apply (blast dest: merK_neq_priEK)
done


lemma Key_analz_image_Key_lemma:
     "P --> (Key K ∈ analz (Key`KK Un H)) --> (K∈KK | Key K ∈ analz H)
      ==>
      P --> (Key K ∈ analz (Key`KK Un H)) = (K∈KK | Key K ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])

lemma symKey_compromise:
     "evs ∈ set_mr ==>
      (∀SK KK. SK ∈ symKeys ⟶ (∀K ∈ KK. K ∉ range(λC. priEK C)) -->
               (Key SK ∈ analz (Key`KK Un (knows Spy evs))) =
               (SK ∈ KK | Key SK ∈ analz (knows Spy evs)))"
apply (erule set_mr.induct)
apply (safe del: impI intro!: Key_analz_image_Key_lemma [THEN impI])
apply (drule_tac [7] msg4_priEK_disj)
apply (frule_tac [6] Gets_certificate_valid)
apply (safe del: impI)
apply (simp_all del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps abbrev_simps analz_knows_absorb
              analz_knows_absorb2 analz_Key_image_insert_eq notin_image_iff
              Spy_analz_private_Key analz_image_priEK)
  — 23 seconds on a 1.8GHz machine
apply spy_analz  — Fake
apply auto  — Message 3
done

lemma symKey_secrecy [rule_format]:
     "[|CA i ∉ bad; K ∈ symKeys;  evs ∈ set_mr|]
      ==> ∀X m. Says (Merchant m) (CA i) X ∈ set evs -->
                Key K ∈ parts{X} -->
                Merchant m ∉ bad -->
                Key K ∉ analz (knows Spy evs)"
apply (erule set_mr.induct)
apply (drule_tac [7] msg4_priEK_disj)
apply (frule_tac [6] Gets_certificate_valid)
apply (safe del: impI)
apply (simp_all del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps abbrev_simps analz_knows_absorb
              analz_knows_absorb2 analz_Key_image_insert_eq
              symKey_compromise notin_image_iff Spy_analz_private_Key
              analz_image_priEK)
apply spy_analz  — Fake
apply force  — Message 1
apply (auto intro: analz_into_parts [THEN usedI] in_parts_Says_imp_used)  —
Message 3
done
```

5.2 Unicity

lemma *msg4_Says_imp_Notes:*

```
 "[|Says (CA i) M {|sign (priSK (CA i)) {|Agent M, Nonce NM2, Agent (CA i)|},
                   cert  M      merSK    onlySig (priSK (CA i)),
                   cert  M      merEK    onlyEnc (priSK (CA i)),
                   cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|} ∈ set
evs;
    evs ∈ set_mr |]
  ==> Notes (CA i) (Key merSK) ∈ set evs
   &  Notes (CA i) (Key merEK) ∈ set evs"
apply (erule rev_mp)
apply (erule set_mr.induct)
apply (simp_all (no_asm_simp))
done
```

Unicity of merSK wrt a given CA: merSK uniquely identifies the other components, including merEK

```
lemma merSK_unicity:
 "[|Says (CA i) M {|sign (priSK(CA i)) {|Agent M, Nonce NM2, Agent (CA i)|},
                   cert  M      merSK    onlySig (priSK (CA i)),
                   cert  M      merEK    onlyEnc (priSK (CA i)),
                   cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|} ∈ set
evs;
    Says (CA i) M' {|sign (priSK(CA i)) {|Agent M', Nonce NM2', Agent (CA i)|},
                   cert  M'      merSK    onlySig (priSK (CA i)),
                   cert  M'      merEK'    onlyEnc (priSK (CA i)),
                   cert (CA i) (pubSK(CA i)) onlySig (priSK RCA)|} ∈ set
evs;
    evs ∈ set_mr |] ==> M=M' & NM2=NM2' & merEK=merEK'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_mr.induct)
apply (simp_all (no_asm_simp))
apply (blast dest!: msg4_Says_imp_Notes)
done
```

Unicity of merEK wrt a given CA: merEK uniquely identifies the other components, including merSK

```
lemma merEK_unicity:
 "[|Says (CA i) M {|sign (priSK(CA i)) {|Agent M, Nonce NM2, Agent (CA i)|},
                   cert  M      merSK    onlySig (priSK (CA i)),
                   cert  M      merEK    onlyEnc (priSK (CA i)),
                   cert (CA i) (pubSK (CA i)) onlySig (priSK RCA)|} ∈ set
evs;
    Says (CA i) M' {|sign (priSK(CA i)) {|Agent M', Nonce NM2', Agent (CA i)|},
                    cert  M'      merSK'    onlySig (priSK (CA i)),
                    cert  M'      merEK    onlyEnc (priSK (CA i)),
                    cert (CA i) (pubSK(CA i)) onlySig (priSK RCA)|} ∈ set
evs;
    evs ∈ set_mr |]
  ==> M=M' & NM2=NM2' & merSK=merSK'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_mr.induct)
apply (simp_all (no_asm_simp))
```

```
apply (blast dest!: msg4_Says_imp_Notes)
done
```

-No interest on secrecy of nonces: they appear to be used only for freshness. -No interest on secrecy of merSK or merEK, as in CR. -There's no equivalent of the PAN

5.3 Primary Goals of Merchant Registration

5.3.1 The merchant's certificates really were created by the CA, provided the CA is uncompromised

The assumption *CA i ≠ RCA* is required: step 2 uses certificates of the same form.

```
lemma certificate_merSK_valid_lemma [intro]:
     "[|Crypt (priSK (CA i)) {|Agent M, Key merSK, onlySig|}
          ∈ parts (knows Spy evs);
        CA i ∉ bad;  CA i ≠ RCA;  evs ∈ set_mr|]
 ==> ∃X Y Z. Says (CA i) M
                  {|X, cert M merSK onlySig (priSK (CA i)), Y, Z|} ∈ set evs"
apply (erule rev_mp)
apply (erule set_mr.induct)
apply (simp_all (no_asm_simp))
apply auto
done

lemma certificate_merSK_valid:
     "[| cert M merSK onlySig (priSK (CA i)) ∈ parts (knows Spy evs);
         CA i ∉ bad;  CA i ≠ RCA;  evs ∈ set_mr|]
 ==> ∃X Y Z. Says (CA i) M
                  {|X, cert M merSK onlySig (priSK (CA i)), Y, Z|} ∈ set evs"
by auto

lemma certificate_merEK_valid_lemma [intro]:
     "[|Crypt (priSK (CA i)) {|Agent M, Key merEK, onlyEnc|}
          ∈ parts (knows Spy evs);
        CA i ∉ bad;  CA i ≠ RCA;  evs ∈ set_mr|]
 ==> ∃X Y Z. Says (CA i) M
                  {|X, Y, cert M merEK onlyEnc (priSK (CA i)), Z|} ∈ set evs"
apply (erule rev_mp)
apply (erule set_mr.induct)
apply (simp_all (no_asm_simp))
apply auto
done

lemma certificate_merEK_valid:
     "[| cert M merEK onlyEnc (priSK (CA i)) ∈ parts (knows Spy evs);
         CA i ∉ bad;  CA i ≠ RCA;  evs ∈ set_mr|]
 ==> ∃X Y Z. Says (CA i) M
                  {|X, Y, cert M merEK onlyEnc (priSK (CA i)), Z|} ∈ set evs"
by auto
```

The two certificates - for merSK and for merEK - cannot be proved to have originated together

end

6 Purchase Phase of SET

theory *Purchase = PublicSET:*

Note: nonces seem to consist of 20 bytes. That includes both freshness challenges (Chall-EE, etc.) and important secrets (CardSecret, PANsecret)

This version omits *LID_C* but retains *LID_M*. At first glance (Programmer's Guide page 267) it seems that both numbers are just introduced for the respective convenience of the Cardholder's and Merchant's system. However, omitting both of them would create a problem of identification: how can the Merchant's system know what transaction is it supposed to process?

Further reading (Programmer's guide page 309) suggest that there is an outside bootstrapping message (SET initiation message) which is used by the Merchant and the Cardholder to agree on the actual transaction. This bootstrapping message is described in the SET External Interface Guide and ought to generate *LID_M*. According SET Extern Interface Guide, this number might be a cookie, an invoice number etc. The Programmer's Guide on page 310, states that in absence of *LID_M* the protocol must somehow ("outside SET") identify the transaction from OrderDesc, which is assumed to be a searchable text only field. Thus, it is assumed that the Merchant or the Cardholder somehow agreed out-of-bad on the value of *LID_M* (for instance a cookie in a web transaction etc.). This out-of-band agreement is expressed with a preliminary start action in which the merchant and the Cardholder agree on the appropriate values. Agreed values are stored with a suitable notes action.

"XID is a transaction ID that is usually generated by the Merchant system, unless there is no PInitRes, in which case it is generated by the Cardholder system. It is a randomly generated 20 byte variable that is globally unique (statistically). Merchant and Cardholder systems shall use appropriate random number generators to ensure the global uniqueness of XID." –Programmer's Guide, page 267.

PI (Payment Instruction) is the most central and sensitive data structure in SET. It is used to pass the data required to authorize a payment card payment from the Cardholder to the Payment Gateway, which will use the data to initiate a payment card transaction through the traditional payment card financial network. The data is encrypted by the Cardholder and sent via the Merchant, such that the data is hidden from the Merchant unless the Acquirer passes the data back to the Merchant. –Programmer's Guide, page 271.

consts

```
CardSecret :: "nat => nat"


PANSecret :: "nat => nat"


set_pur  :: "event list set"
```

inductive *set_pur*
intros

Nil: — Initial trace is empty
```
"[] ∈ set_pur"
```

Fake: — The spy MAY say anything he CAN say.
```
"[| evsf ∈ set_pur;  X ∈ synth(analz(knows Spy evsf)) |]
 ==> Says Spy B X  # evsf ∈ set_pur"
```

Reception: — If A sends a message X to B, then B might receive it
```
"[| evsr ∈ set_pur;  Says A B X ∈ set evsr |]
 ==> Gets B X  # evsr ∈ set_pur"
```

Start:
— Added start event which is out-of-band for SET: the Cardholder and the merchant agree on the amounts and uses *LID_M* as an identifier. This is suggested by the External Interface Guide. The Programmer's Guide, in absence of *LID_M*, states that the merchant uniquely identifies the order out of some data contained in OrderDesc.
```
"[|evsStart ∈ set_pur;
   Number LID_M ∉ used evsStart;
   C = Cardholder k; M = Merchant i; P = PG j;
   Transaction = {|Agent M, Agent C, Number OrderDesc, Number PurchAmt|};
   LID_M ∉ range CardSecret;
   LID_M ∉ range PANSecret |]
  ==> Notes C {|Number LID_M, Transaction|}
    # Notes M {|Number LID_M, Agent P, Transaction|}
    # evsStart ∈ set_pur"
```

PInitReq:
— Purchase initialization, page 72 of Formal Protocol Desc.
```
"[|evsPIReq ∈ set_pur;
   Transaction = {|Agent M, Agent C, Number OrderDesc, Number PurchAmt|};
   Nonce Chall_C ∉ used evsPIReq;
   Chall_C ∉ range CardSecret; Chall_C ∉ range PANSecret;
   Notes C {|Number LID_M, Transaction |} ∈ set evsPIReq |]
 ==> Says C M {|Number LID_M, Nonce Chall_C|} # evsPIReq ∈ set_pur"
```

PInitRes:
— Merchant replies with his own label XID and the encryption key certificate of his chosen Payment Gateway. Page 74 of Formal Protocol Desc. We use *LID_M* to identify Cardholder
```
"[|evsPIRes ∈ set_pur;
   Gets M {|Number LID_M, Nonce Chall_C|} ∈ set evsPIRes;
   Transaction = {|Agent M, Agent C, Number OrderDesc, Number PurchAmt|};
   Notes M {|Number LID_M, Agent P, Transaction|} ∈ set evsPIRes;
   Nonce Chall_M ∉ used evsPIRes;
   Chall_M ∉ range CardSecret; Chall_M ∉ range PANSecret;
   Number XID ∉ used evsPIRes;
   XID ∉ range CardSecret; XID ∉ range PANSecret|]
 ==> Says M C (sign (priSK M)
```

```
                   {|Number LID_M, Number XID,
                     Nonce Chall_C, Nonce Chall_M,
                     cert P (pubEK P) onlyEnc (priSK RCA)|})
        # evsPIRes ∈ set_pur"
```

PReqUns:

— UNSIGNED Purchase request (CardSecret = 0). Page 79 of Formal Protocol Desc. Merchant never sees the amount in clear. This holds of the real protocol, where XID identifies the transaction. We omit Hash—Number XID, Nonce (CardSecret k)— from PIHead because the CardSecret is 0 and because AuthReq treated the unsigned case very differently from the signed one anyway.

```
"[|evsPReqU ∈ set_pur;
   C = Cardholder k; CardSecret k = 0;
   Key KC1 ∉ used evsPReqU;  KC1 ∈ symKeys;
   Transaction = {|Agent M, Agent C, Number OrderDesc, Number PurchAmt|};
   HOD = Hash{|Number OrderDesc, Number PurchAmt|};
   OIData = {|Number LID_M, Number XID, Nonce Chall_C, HOD,Nonce Chall_M|};
   PIHead = {|Number LID_M, Number XID, HOD, Number PurchAmt, Agent M|};
   Gets C (sign (priSK M)
               {|Number LID_M, Number XID,
                 Nonce Chall_C, Nonce Chall_M,
                 cert P EKj onlyEnc (priSK RCA)|})
     ∈ set evsPReqU;
   Says C M {|Number LID_M, Nonce Chall_C|} ∈ set evsPReqU;
   Notes C {|Number LID_M, Transaction|} ∈ set evsPReqU |]
 ==> Says C M
         {|EXHcrypt KC1 EKj {|PIHead, Hash OIData|} (Pan (pan C)),
           OIData, Hash{|PIHead, Pan (pan C)|} |}
       # Notes C {|Key KC1, Agent M|}
       # evsPReqU ∈ set_pur"
```

PReqS:

— SIGNED Purchase request. Page 77 of Formal Protocol Desc. We could specify the equation *PIReqSigned = {|PIDualSigned, OIDualSigned|}*, since the Formal Desc. gives PIHead the same format in the unsigned case. However, there's little point, as P treats the signed and unsigned cases differently.

```
"[|evsPReqS ∈ set_pur;
   C = Cardholder k;
   CardSecret k ≠ 0;  Key KC2 ∉ used evsPReqS;  KC2 ∈ symKeys;
   Transaction = {|Agent M, Agent C, Number OrderDesc, Number PurchAmt|};
   HOD = Hash{|Number OrderDesc, Number PurchAmt|};
   OIData = {|Number LID_M, Number XID, Nonce Chall_C, HOD, Nonce Chall_M|};
   PIHead = {|Number LID_M, Number XID, HOD, Number PurchAmt, Agent M,
              Hash{|Number XID, Nonce (CardSecret k)|}|};
   PANData = {|Pan (pan C), Nonce (PANSecret k)|};
   PIData = {|PIHead, PANData|};
   PIDualSigned = {|sign (priSK C) {|Hash PIData, Hash OIData|},
                    EXcrypt KC2 EKj {|PIHead, Hash OIData|} PANData|};
   OIDualSigned = {|OIData, Hash PIData|};
   Gets C (sign (priSK M)
               {|Number LID_M, Number XID,
                 Nonce Chall_C, Nonce Chall_M,
                 cert P EKj onlyEnc (priSK RCA)|})
     ∈ set evsPReqS;
```

```
    Says C M {|Number LID_M, Nonce Chall_C|} ∈ set evsPReqS;
    Notes C {|Number LID_M, Transaction|} ∈ set evsPReqS |]
  ==> Says C M {|PIDualSigned, OIDualSigned|}
        # Notes C {|Key KC2, Agent M|}
        # evsPReqS ∈ set_pur"
```

— Authorization Request. Page 92 of Formal Protocol Desc. Sent in response to Purchase Request.

```
AuthReq:
 "[| evsAReq ∈ set_pur;
     Key KM ∉ used evsAReq;  KM ∈ symKeys;
     Transaction = {|Agent M, Agent C, Number OrderDesc, Number PurchAmt|};
     HOD = Hash{|Number OrderDesc, Number PurchAmt|};
     OIData = {|Number LID_M, Number XID, Nonce Chall_C, HOD,
                Nonce Chall_M|};
     CardSecret k ≠ 0 -->
       P_I = {|sign (priSK C) {|HPIData, Hash OIData|}, encPANData|};
     Gets M {|P_I, OIData, HPIData|} ∈ set evsAReq;
     Says M C (sign (priSK M) {|Number LID_M, Number XID,
                                Nonce Chall_C, Nonce Chall_M,
                                cert P EKj onlyEnc (priSK RCA)|})
       ∈ set evsAReq;
      Notes M {|Number LID_M, Agent P, Transaction|}
         ∈ set evsAReq |]
  ==> Says M P
          (EncB (priSK M) KM (pubEK P)
            {|Number LID_M, Number XID, Hash OIData, HOD|}   P_I)
        # evsAReq ∈ set_pur"
```

— Authorization Response has two forms: for UNSIGNED and SIGNED PIs. Page 99 of Formal Protocol Desc. PI is a keyword (product!), so we call it *P_I*. The hashes HOD and HOIData occur independently in *P_I* and in M's message. The authCode in AuthRes represents the baggage of EncB, which in the full protocol is [CapToken], [AcqCardMsg], [AuthToken]: optional items for split shipments, recurring payments, etc.

```
AuthResUns:
```

— Authorization Response, UNSIGNED

```
 "[| evsAResU ∈ set_pur;
     C = Cardholder k; M = Merchant i;
     Key KP ∉ used evsAResU;  KP ∈ symKeys;
     CardSecret k = 0;  KC1 ∈ symKeys;  KM ∈ symKeys;
     PIHead = {|Number LID_M, Number XID, HOD, Number PurchAmt, Agent M|};
     P_I = EXHcrypt KC1 EKj {|PIHead, HOIData|} (Pan (pan C));
     Gets P (EncB (priSK M) KM (pubEK P)
             {|Number LID_M, Number XID, HOIData, HOD|} P_I)
         ∈ set evsAResU |]
 ==> Says P M
          (EncB (priSK P) KP (pubEK M)
            {|Number LID_M, Number XID, Number PurchAmt|}
            authCode)
     # evsAResU ∈ set_pur"

AuthResS:
```

— Authorization Response, SIGNED

```
"[| evsAResS ∈ set_pur;
    C = Cardholder k;
    Key KP ∉ used evsAResS;  KP ∈ symKeys;
    CardSecret k ≠ 0;  KC2 ∈ symKeys;  KM ∈ symKeys;
    P_I = {|sign (priSK C) {|Hash PIData, HOIData|},
            EXcrypt KC2 (pubEK P) {|PIHead, HOIData|} PANData|};
    PANData = {|Pan (pan C), Nonce (PANSecret k)|};
    PIData = {|PIHead, PANData|};
    PIHead = {|Number LID_M, Number XID, HOD, Number PurchAmt, Agent M,
               Hash{|Number XID, Nonce (CardSecret k)|}|};
    Gets P (EncB (priSK M) KM (pubEK P)
             {|Number LID_M, Number XID, HOIData, HOD|}
            P_I)
        ∈ set evsAResS |]
 ==> Says P M
          (EncB (priSK P) KP (pubEK M)
            {|Number LID_M, Number XID, Number PurchAmt|}
            authCode)
    # evsAResS ∈ set_pur"
```

PRes:

— Purchase response.

```
"[| evsPRes ∈ set_pur;  KP ∈ symKeys;  M = Merchant i;
    Transaction = {|Agent M, Agent C, Number OrderDesc, Number PurchAmt|};
    Gets M (EncB (priSK P) KP (pubEK M)
            {|Number LID_M, Number XID, Number PurchAmt|}
            authCode)
       ∈ set evsPRes;
    Gets M {|Number LID_M, Nonce Chall_C|} ∈ set evsPRes;
    Says M P
         (EncB (priSK M) KM (pubEK P)
           {|Number LID_M, Number XID, Hash OIData, HOD|} P_I)
      ∈ set evsPRes;
    Notes M {|Number LID_M, Agent P, Transaction|}
       ∈ set evsPRes
   |]
 ==> Says M C
      (sign (priSK M) {|Number LID_M, Number XID, Nonce Chall_C,
                        Hash (Number PurchAmt)|})
      # evsPRes ∈ set_pur"
```

specification *(CardSecret PANSecret)*
  inj_CardSecret: "inj CardSecret"
  inj_PANSecret: "inj PANSecret"
  CardSecret_neq_PANSecret: "CardSecret k ≠ PANSecret k'"
  — No CardSecret equals any PANSecret
  apply *(rule_tac x="curry nat2_to_nat 0"* **in** *exI)*
  apply *(rule_tac x="curry nat2_to_nat 1"* **in** *exI)*
  apply *(simp add: nat2_to_nat_inj [THEN inj_eq] inj_on_def)*
  done

declare *Says_imp_knows_Spy [THEN parts.Inj, dest]*

```
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

declare CardSecret_neq_PANSecret [iff]
        CardSecret_neq_PANSecret [THEN not_sym, iff]
declare inj_CardSecret [THEN inj_eq, iff]
        inj_PANSecret [THEN inj_eq, iff]
```

6.1 Possibility Properties

```
lemma Says_to_Gets:
     "Says A B X # evs ∈ set_pur ==> Gets B X # Says A B X # evs ∈ set_pur"
by (rule set_pur.Reception, auto)
```

Possibility for UNSIGNED purchases. Note that we need to ensure that XID differs from OrderDesc and PurchAmt, since it is supposed to be a unique number!

```
lemma possibility_Uns:
    "[| CardSecret k = 0;
        C = Cardholder k;  M = Merchant i;
        Key KC ∉ used []; Key KM ∉ used []; Key KP ∉ used [];
        KC ∈ symKeys; KM ∈ symKeys; KP ∈ symKeys;
        KC < KM; KM < KP;
        Nonce Chall_C ∉ used []; Chall_C ∉ range CardSecret ∪ range PANSecret;
        Nonce Chall_M ∉ used []; Chall_M ∉ range CardSecret ∪ range PANSecret;
        Chall_C < Chall_M;
        Number LID_M ∉ used []; LID_M ∉ range CardSecret ∪ range PANSecret;
        Number XID ∉ used []; XID ∉ range CardSecret ∪ range PANSecret;
        LID_M < XID; XID < OrderDesc; OrderDesc < PurchAmt |]
   ==> ∃evs ∈ set_pur.
          Says M C
               (sign (priSK M)
                    {|Number LID_M, Number XID, Nonce Chall_C,
                      Hash (Number PurchAmt)|})
                  ∈ set evs"
apply (intro exI bexI)
apply (rule_tac [2]
        set_pur.Nil
         [THEN set_pur.Start [of _ LID_M C k M i _ _ _ OrderDesc PurchAmt],

          THEN set_pur.PInitReq [of concl: C M LID_M Chall_C],
          THEN Says_to_Gets,
          THEN set_pur.PInitRes [of concl: M C LID_M XID Chall_C Chall_M],

          THEN Says_to_Gets,
          THEN set_pur.PReqUns [of concl: C M KC],
          THEN Says_to_Gets,
          THEN set_pur.AuthReq [of concl: M "PG j" KM LID_M XID],
          THEN Says_to_Gets,
          THEN set_pur.AuthResUns [of concl: "PG j" M KP LID_M XID],
          THEN Says_to_Gets,
          THEN set_pur.PRes])
apply (tactic "basic_possibility_tac")
```

```
apply (simp_all add: used_Cons symKeys_neq_imp_neq)
done

lemma possibility_S:
    "[| CardSecret k ≠ 0;
        C = Cardholder k;  M = Merchant i;
        Key KC ∉ used []; Key KM ∉ used []; Key KP ∉ used [];
        KC ∈ symKeys; KM ∈ symKeys; KP ∈ symKeys;
        KC < KM; KM < KP;
        Nonce Chall_C ∉ used []; Chall_C ∉ range CardSecret ∪ range PANSecret;
        Nonce Chall_M ∉ used []; Chall_M ∉ range CardSecret ∪ range PANSecret;
        Chall_C < Chall_M;
        Number LID_M ∉ used []; LID_M ∉ range CardSecret ∪ range PANSecret;
        Number XID ∉ used []; XID ∉ range CardSecret ∪ range PANSecret;
        LID_M < XID; XID < OrderDesc; OrderDesc < PurchAmt |]
   ==>  ∃evs ∈ set_pur.
            Says M C
                 (sign (priSK M) {|Number LID_M, Number XID, Nonce Chall_C,

                                   Hash (Number PurchAmt)|})
              ∈ set evs"
apply (intro exI bexI)
apply (rule_tac [2]
        set_pur.Nil
         [THEN set_pur.Start [of _ LID_M C k M i _ _ _ OrderDesc PurchAmt],

          THEN set_pur.PInitReq [of concl: C M LID_M Chall_C],
          THEN Says_to_Gets,
          THEN set_pur.PInitRes [of concl: M C LID_M XID Chall_C Chall_M],

          THEN Says_to_Gets,
          THEN set_pur.PReqS [of concl: C M _ _ KC],
          THEN Says_to_Gets,
          THEN set_pur.AuthReq [of concl: M "PG j" KM LID_M XID],
          THEN Says_to_Gets,
          THEN set_pur.AuthResS [of concl: "PG j" M KP LID_M XID],
          THEN Says_to_Gets,
          THEN set_pur.PRes])
apply (tactic "basic_possibility_tac")
apply (auto simp add: used_Cons symKeys_neq_imp_neq)
done


lemma Gets_imp_Says:
     "[| Gets B X ∈ set evs; evs ∈ set_pur |]
   ==> ∃A. Says A B X ∈ set evs"
apply (erule rev_mp)
apply (erule set_pur.induct, auto)
done

lemma Gets_imp_knows_Spy:
     "[| Gets B X ∈ set evs; evs ∈ set_pur |]  ==> X ∈ knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)
```

```
declare Gets_imp_knows_Spy [THEN parts.Inj, dest]
```

Forwarding lemmas, to aid simplification

```
lemma AuthReq_msg_in_parts_spies:
     "[|Gets M {|P_I, OIData, HPIData|} ∈ set evs;
        evs ∈ set_pur|] ==> P_I ∈ parts (knows Spy evs)"
by auto

lemma AuthReq_msg_in_analz_spies:
     "[|Gets M {|P_I, OIData, HPIData|} ∈ set evs;
        evs ∈ set_pur|] ==> P_I ∈ analz (knows Spy evs)"
by (blast dest: Gets_imp_knows_Spy [THEN analz.Inj])
```

6.2 Proofs on Asymmetric Keys

Private Keys are Secret

Spy never sees an agent's private keys! (unless it's bad at start)

```
lemma Spy_see_private_Key [simp]:
     "evs ∈ set_pur
      ==> (Key(invKey (publicKey b A)) ∈ parts(knows Spy evs)) = (A ∈ bad)"
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply auto
done
declare Spy_see_private_Key [THEN [2] rev_iffD1, dest!]

lemma Spy_analz_private_Key [simp]:
     "evs ∈ set_pur ==>
     (Key(invKey (publicKey b A)) ∈ analz(knows Spy evs)) = (A ∈ bad)"
by auto
declare Spy_analz_private_Key [THEN [2] rev_iffD1, dest!]
```

rewriting rule for priEK's

```
lemma parts_image_priEK:
     "[|Key (priEK C) ∈ parts (Key`KK Un (knows Spy evs));
        evs ∈ set_pur|] ==> priEK C ∈ KK | C ∈ bad"
by auto
```

trivial proof because (priEK C) never appears even in (parts evs)

```
lemma analz_image_priEK:
     "evs ∈ set_pur ==>
          (Key (priEK C) ∈ analz (Key`KK Un (knows Spy evs))) =
          (priEK C ∈ KK | C ∈ bad)"
by (blast dest!: parts_image_priEK intro: analz_mono [THEN [2] rev_subsetD])
```

6.3 Public Keys in Certificates are Correct

```
lemma Crypt_valid_pubEK [dest!]:
     "[| Crypt (priSK RCA) {|Agent C, Key EKi, onlyEnc|}
           ∈ parts (knows Spy evs);
         evs ∈ set_pur |] ==> EKi = pubEK C"
apply (erule rev_mp)
```

apply *(erule set_pur.induct, auto)*
done

```
lemma Crypt_valid_pubSK [dest!]:
     "[| Crypt (priSK RCA) {|Agent C, Key SKi, onlySig|}
           ∈ parts (knows Spy evs);
         evs ∈ set_pur |] ==> SKi = pubSK C"
apply (erule rev_mp)
apply (erule set_pur.induct, auto)
done

lemma certificate_valid_pubEK:
    "[| cert C EKi onlyEnc (priSK RCA) ∈ parts (knows Spy evs);
        evs ∈ set_pur |]
     ==> EKi = pubEK C"
by (unfold cert_def signCert_def, auto)

lemma certificate_valid_pubSK:
    "[| cert C SKi onlySig (priSK RCA) ∈ parts (knows Spy evs);
        evs ∈ set_pur |] ==> SKi = pubSK C"
by (unfold cert_def signCert_def, auto)

lemma Says_certificate_valid [simp]:
     "[| Says A B (sign SK {|lid, xid, cc, cm,
                           cert C EK onlyEnc (priSK RCA)|}) ∈ set evs;
         evs ∈ set_pur |]
      ==> EK = pubEK C"
by (unfold sign_def, auto)

lemma Gets_certificate_valid [simp]:
     "[| Gets A (sign SK {|lid, xid, cc, cm,
                           cert C EK onlyEnc (priSK RCA)|}) ∈ set evs;
         evs ∈ set_pur |]
      ==> EK = pubEK C"
by (frule Gets_imp_Says, auto)

ML
{*
val Gets_certificate_valid = thm "Gets_certificate_valid";

fun valid_certificate_tac i =
    EVERY [ftac Gets_certificate_valid i,
           assume_tac i, REPEAT (hyp_subst_tac i)];
*}
```

6.4 Proofs on Symmetric Keys

Nobody can have used non-existent keys!

```
lemma new_keys_not_used [rule_format,simp]:
     "evs ∈ set_pur
      ==> Key K ∉ used evs --> K ∈ symKeys -->
          K ∉ keysFor (parts (knows Spy evs))"
apply (erule set_pur.induct)
```

```
apply (tactic{*valid_certificate_tac 8*}) — PReqS
apply (tactic{*valid_certificate_tac 7*}) — PReqUns
apply auto
apply (force dest!: usedI keysFor_parts_insert) — Fake
done

lemma new_keys_not_analzd:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_pur |]
      ==> K ∉ keysFor (analz (knows Spy evs))"
by (blast intro: keysFor_mono [THEN [2] rev_subsetD] dest: new_keys_not_used)

lemma Crypt_parts_imp_used:
     "[|Crypt K X ∈ parts (knows Spy evs);
        K ∈ symKeys; evs ∈ set_pur |] ==> Key K ∈ used evs"
apply (rule ccontr)
apply (force dest: new_keys_not_used Crypt_imp_invKey_keysFor)
done

lemma Crypt_analz_imp_used:
     "[|Crypt K X ∈ analz (knows Spy evs);
        K ∈ symKeys; evs ∈ set_pur |] ==> Key K ∈ used evs"
by (blast intro: Crypt_parts_imp_used)
```

New versions: as above, but generalized to have the KK argument

```
lemma gen_new_keys_not_used:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_pur |]
      ==> Key K ∉ used evs --> K ∈ symKeys -->
          K ∉ keysFor (parts (Key`KK Un knows Spy evs))"
by auto

lemma gen_new_keys_not_analzd:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_pur |]
      ==> K ∉ keysFor (analz (Key`KK Un knows Spy evs))"
by (blast intro: keysFor_mono [THEN subsetD] dest: gen_new_keys_not_used)

lemma analz_Key_image_insert_eq:
     "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ set_pur |]
      ==> analz (Key ` (insert K KK) ∪ knows Spy evs) =
          insert (Key K) (analz (Key ` KK ∪ knows Spy evs))"
by (simp add: gen_new_keys_not_analzd)
```

6.5 Secrecy of Symmetric Keys

```
lemma Key_analz_image_Key_lemma:
     "P --> (Key K ∈ analz (Key`KK Un H)) --> (K∈KK | Key K ∈ analz H)
      ==>
      P --> (Key K ∈ analz (Key`KK Un H)) = (K∈KK | Key K ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])


lemma symKey_compromise:
     "evs ∈ set_pur ==>
      (∀SK KK. SK ∈ symKeys ⟶
        (∀K ∈ KK. K ∉ range(λC. priEK C)) -->
```

```
              (Key SK ∈ analz (Key`KK Un (knows Spy evs))) =
              (SK ∈ KK | Key SK ∈ analz (knows Spy evs)))"
apply (erule set_pur.induct)
apply (rule_tac [!] allI)+
apply (rule_tac [!] impI [THEN Key_analz_image_Key_lemma, THEN impI])+
apply (frule_tac [9] AuthReq_msg_in_analz_spies) — AReq
apply (tactic{*valid_certificate_tac 8*}) — PReqS
apply (tactic{*valid_certificate_tac 7*}) — PReqUns
apply (simp_all
         del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps disj_simps
              analz_Key_image_insert_eq notin_image_iff
              analz_insert_simps analz_image_priEK)
  — 35 seconds on a 1.8GHz machine
apply spy_analz — Fake
apply (blast elim!: ballE)+ — PReq: unsigned and signed
done
```

6.6 Secrecy of Nonces

As usual: we express the property as a logical equivalence

```
lemma Nonce_analz_image_Key_lemma:
     "P --> (Nonce N ∈ analz (Key`KK Un H)) --> (Nonce N ∈ analz H)
      ==> P --> (Nonce N ∈ analz (Key`KK Un H)) = (Nonce N ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])
```

The (no_asm) attribute is essential, since it retains the quantifier and allows the simprule's condition to itself be simplified.

```
lemma Nonce_compromise [rule_format (no_asm)]:
     "evs ∈ set_pur ==>
      (∀N KK. (∀K ∈ KK. K ∉ range(λC. priEK C))   -->
              (Nonce N ∈ analz (Key`KK Un (knows Spy evs))) =
              (Nonce N ∈ analz (knows Spy evs)))"
apply (erule set_pur.induct)
apply (rule_tac [!] allI)+
apply (rule_tac [!] impI [THEN Nonce_analz_image_Key_lemma])+
apply (frule_tac [9] AuthReq_msg_in_analz_spies) — AReq
apply (tactic{*valid_certificate_tac 8*}) — PReqS
apply (tactic{*valid_certificate_tac 7*}) — PReqUns
apply (simp_all
         del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps disj_simps symKey_compromise
              analz_Key_image_insert_eq notin_image_iff
              analz_insert_simps analz_image_priEK)
  — 35 seconds on a 1.8GHz machine
apply spy_analz — Fake
apply (blast elim!: ballE) — PReqS
done
```

```
lemma PANSecret_notin_spies:
     "evs ∈ set_pur
   ==> Nonce (PANSecret k) ∈ analz (knows Spy evs) -->
       (∃V W X Y KC2 M. ∃P ∈ bad.
```

```
        Says (Cardholder k) M
             {|{|W, EXcrypt KC2 (pubEK P) X {|Y, Nonce (PANSecret k)|}|},
               V|} ∈ set evs)"
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_analz_spies)
apply (tactic{*valid_certificate_tac 8*}) — PReqS
apply (simp_all
         del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps disj_simps
              symKey_compromise pushes sign_def Nonce_compromise
              analz_Key_image_insert_eq notin_image_iff
              analz_insert_simps analz_image_priEK)
  — 13 seconds on a 1.8GHz machine
apply spy_analz
apply (blast dest!: Gets_imp_knows_Spy [THEN analz.Inj])
apply (blast dest: Says_imp_knows_Spy [THEN analz.Inj]
                   Gets_imp_knows_Spy [THEN analz.Inj])
apply (blast dest: Gets_imp_knows_Spy [THEN analz.Inj]) — PReqS
apply (blast dest: Says_imp_knows_Spy [THEN analz.Inj]
                   Gets_imp_knows_Spy [THEN analz.Inj]) — PRes
done
```

This theorem is a bit silly, in that many CardSecrets are 0! But then we don't care. NOT USED

```
lemma CardSecret_notin_spies:
     "evs ∈ set_pur ==> Nonce (CardSecret i) ∉ parts (knows Spy evs)"
by (erule set_pur.induct, auto)
```

6.7 Confidentiality of PAN

```
lemma analz_image_pan_lemma:
     "(Pan P ∈ analz (Key`nE Un H)) --> (Pan P ∈ analz H)  ==>
      (Pan P ∈ analz (Key`nE Un H)) =   (Pan P ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])
```

The (no_asm) attribute is essential, since it retains the quantifier and allows the simprule's condition to itself be simplified.

```
lemma analz_image_pan [rule_format (no_asm)]:
     "evs ∈ set_pur ==>
       ∀KK. (∀K ∈ KK. K ∉ range(λC. priEK C)) -->
            (Pan P ∈ analz (Key`KK Un (knows Spy evs))) =
            (Pan P ∈ analz (knows Spy evs))"
apply (erule set_pur.induct)
apply (rule_tac [!] allI impI)+
apply (rule_tac [!] analz_image_pan_lemma)+
apply (frule_tac [9] AuthReq_msg_in_analz_spies) — AReq
apply (tactic{*valid_certificate_tac 8*}) — PReqS
apply (tactic{*valid_certificate_tac 7*}) — PReqUns
apply (simp_all
         del: image_insert image_Un imp_disjL
         add: analz_image_keys_simps
              symKey_compromise pushes sign_def
              analz_Key_image_insert_eq notin_image_iff
```

```
             analz_insert_simps analz_image_priEK)
```
— 32 seconds on a 1.8GHz machine

apply *spy_analz* — Fake
apply *auto*
done

```
lemma analz_insert_pan:
     "[| evs ∈ set_pur;  K ∉ range(λC. priEK C) |] ==>
          (Pan P ∈ analz (insert (Key K) (knows Spy evs))) =
          (Pan P ∈ analz (knows Spy evs))"
by (simp del: image_insert image_Un
        add: analz_image_keys_simps analz_image_pan)
```

Confidentiality of the PAN, unsigned case.

```
lemma pan_confidentiality_unsigned:
     "[| Pan (pan C) ∈ analz(knows Spy evs);  C = Cardholder k;
         CardSecret k = 0;  evs ∈ set_pur|]
    ==> ∃P M KC1 K X Y.
     Says C M {|EXHcrypt KC1 (pubEK P) X (Pan (pan C)), Y|}
          ∈ set evs  &
     P ∈ bad"
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_analz_spies) — AReq
apply (tactic{*valid_certificate_tac 8*}) — PReqS
apply (tactic{*valid_certificate_tac 7*}) — PReqUns
apply (simp_all
        del: image_insert image_Un imp_disjL
        add: analz_image_keys_simps analz_insert_pan analz_image_pan
             notin_image_iff
             analz_insert_simps analz_image_priEK)
```
— 15 seconds on a 1.8GHz machine

apply *spy_analz* — Fake
apply *blast* — PReqUns: unsigned
apply *force* — PReqS: signed
done

Confidentiality of the PAN, signed case.

```
lemma pan_confidentiality_signed:
     "[| Pan (pan C) ∈ analz(knows Spy evs);  C = Cardholder k;
         CardSecret k ≠ 0;  evs ∈ set_pur|]
    ==> ∃B M KC2 K ps X Y Z.
     Says C M {|{|X, EXcrypt KC2 (pubEK B) Y {|Pan (pan C), ps|}|}, Z|}
          ∈ set evs  &
     B ∈ bad"
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_analz_spies) — AReq
apply (tactic{*valid_certificate_tac 8*}) — PReqS
apply (tactic{*valid_certificate_tac 7*}) — PReqUns
apply (simp_all
        del: image_insert image_Un imp_disjL
        add: analz_image_keys_simps analz_insert_pan analz_image_pan
             notin_image_iff
```

```
             analz_insert_simps analz_image_priEK)
```
— 17 seconds on a 1.8GHz machine

apply *spy_analz* — Fake
apply *force* — PReqUns: unsigned
apply *blast* — PReqS: signed
done

General goal: that C, M and PG agree on those details of the transaction that they are allowed to know about. PG knows about price and account details. M knows about the order description and price. C knows both.

6.8 Proofs Common to Signed and Unsigned Versions

```
lemma M_Notes_PG:
     "[|Notes M {|Number LID_M, Agent P, Agent M, Agent C, etc|} ∈ set evs;
        evs ∈ set_pur|] ==> ∃j. P = PG j"
apply (erule rev_mp)
apply (erule set_pur.induct, simp_all)
done
```

If we trust M, then *LID_M* determines his choice of P (Payment Gateway)

```
lemma goodM_gives_correct_PG:
     "[| MsgPInitRes =
            {|Number LID_M, xid, cc, cm, cert P EKj onlyEnc (priSK RCA)|};
         Crypt (priSK M) (Hash MsgPInitRes) ∈ parts (knows Spy evs);
         evs ∈ set_pur; M ∉ bad |]
      ==> ∃j trans.
            P = PG j &
            Notes M {|Number LID_M, Agent P, trans|} ∈ set evs"
apply clarify
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply simp_all
apply (blast intro: M_Notes_PG)+
done
```

```
lemma C_gets_correct_PG:
     "[| Gets A (sign (priSK M) {|Number LID_M, xid, cc, cm,
                              cert P EKj onlyEnc (priSK RCA)|}) ∈ set evs;
         evs ∈ set_pur;  M ∉ bad|]
      ==> ∃j trans.
            P = PG j &
            Notes M {|Number LID_M, Agent P, trans|} ∈ set evs &
            EKj = pubEK P"
by (rule refl [THEN goodM_gives_correct_PG, THEN exE], auto)
```

When C receives PInitRes, he learns M's choice of P

```
lemma C_verifies_PInitRes:
 "[| MsgPInitRes = {|Number LID_M, Number XID, Nonce Chall_C, Nonce Chall_M,
           cert P EKj onlyEnc (priSK RCA)|};
     Crypt (priSK M) (Hash MsgPInitRes) ∈ parts (knows Spy evs);
     evs ∈ set_pur;  M ∉ bad|]
```

```
  ==> ∃j trans.
         Notes M {|Number LID_M, Agent P, trans|} ∈ set evs &
         P = PG j &
         EKj = pubEK P"
apply clarify
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply simp_all
apply (blast intro: M_Notes_PG)+
done
```

Corollary of previous one

```
lemma Says_C_PInitRes:
     "[|Says A C (sign (priSK M)
                      {|Number LID_M, Number XID,
                        Nonce Chall_C, Nonce Chall_M,
                        cert P EKj onlyEnc (priSK RCA)|})
           ∈ set evs;  M ∉ bad;  evs ∈ set_pur|]
      ==> ∃j trans.
           Notes M {|Number LID_M, Agent P, trans|} ∈ set evs &
           P = PG j &
           EKj = pubEK (PG j)"
apply (frule Says_certificate_valid)
apply (auto simp add: sign_def)
apply (blast dest: refl [THEN goodM_gives_correct_PG])
apply (blast dest: refl [THEN C_verifies_PInitRes])
done
```

When P receives an AuthReq, he knows that the signed part originated with M. PIRes also has a signed message from M....

```
lemma P_verifies_AuthReq:
     "[| AuthReqData = {|Number LID_M, Number XID, HOIData, HOD|};
         Crypt (priSK M) (Hash {|AuthReqData, Hash P_I|})
           ∈ parts (knows Spy evs);
         evs ∈ set_pur;  M ∉ bad|]
      ==> ∃j trans KM OIData HPIData.
            Notes M {|Number LID_M, Agent (PG j), trans|} ∈ set evs &
            Gets M {|P_I, OIData, HPIData|} ∈ set evs &
            Says M (PG j) (EncB (priSK M) KM (pubEK (PG j)) AuthReqData P_I)
              ∈ set evs"
apply clarify
apply (erule rev_mp)
apply (erule set_pur.induct, simp_all)
apply (frule_tac [4] M_Notes_PG, auto)
done
```

When M receives AuthRes, he knows that P signed it, including the identifying tages and the purchase amount, which he can verify. (Although the spec has SIGNED and UNSIGNED forms of AuthRes, they send the same message to M.)

```
lemma M_verifies_AuthRes:
  "[| MsgAuthRes = {|Number LID_M, Number XID, Number PurchAmt|};
```

```
      Crypt (priSK (PG j)) (Hash MsgAuthRes) ∈ parts (knows Spy evs);
      evs ∈ set_pur;  PG j ∉ bad|]
   ==> ∃i KM HOIData HOD P_I.
        Gets (PG j)
           (EncB (priSK (Merchant i)) KM (pubEK (PG j))
                   {|Number LID_M, Number XID, HOIData, HOD|}
                   P_I) ∈ set evs &
        Says (PG j) (Merchant i)
             (Crypt (pubEK (Merchant i)) (sign (priSK (PG j)) MsgAuthRes))
          ∈ set evs"
apply clarify
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply simp_all
apply blast+
done
```

6.9 Proofs for Unsigned Purchases

What we can derive from the ASSUMPTION that C issued a purchase request. In the unsigned case, we must trust "C": there's no authentication.

```
lemma C_determines_EKj:
     "[| Says C M {|EXHcrypt KC1 EKj {|PIHead, Hash OIData|} (Pan (pan C)),
                   OIData, Hash{|PIHead, Pan (pan C)|} |} ∈ set evs;
         PIHead = {|Number LID_M, Trans_details|};
         evs ∈ set_pur;  C = Cardholder k;  M ∉ bad|]
  ==> ∃trans j.
               Notes M {|Number LID_M, Agent (PG j), trans |} ∈ set evs &
               EKj = pubEK (PG j)"
apply clarify
apply (erule rev_mp)
apply (erule set_pur.induct, simp_all)
apply (tactic{*valid_certificate_tac 2*}) — PReqUns
apply auto
apply (blast dest: Gets_imp_Says Says_C_PInitRes)
done
```

Unicity of *LID_M* between Merchant and Cardholder notes

```
lemma unique_LID_M:
     "[|Notes (Merchant i) {|Number LID_M, Agent P, Trans|} ∈ set evs;
        Notes C {|Number LID_M, Agent M, Agent C, Number OD,
             Number PA|} ∈ set evs;
        evs ∈ set_pur|]
      ==> M = Merchant i & Trans = {|Agent M, Agent C, Number OD, Number PA|}"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_pur.induct, simp_all)
apply (force dest!: Notes_imp_parts_subset_used)
done
```

Unicity of *LID_M*, for two Merchant Notes events

```
lemma unique_LID_M2:
```

```
     "[|Notes M {|Number LID_M, Trans|} ∈ set evs;
        Notes M {|Number LID_M, Trans'|} ∈ set evs;
        evs ∈ set_pur|] ==> Trans' = Trans"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_pur.induct, simp_all)
apply (force dest!: Notes_imp_parts_subset_used)
done
```

Lemma needed below: for the case that if PRes is present, then `LID_M` has been used.

```
lemma signed_imp_used:
     "[| Crypt (priSK M) (Hash X) ∈ parts (knows Spy evs);
         M ∉ bad;  evs ∈ set_pur|] ==> parts {X} ⊆ used evs"
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply simp_all
apply safe
apply blast+
done
```

Similar, with nested Hash

```
lemma signed_Hash_imp_used:
     "[| Crypt (priSK C) (Hash {|H, Hash X|}) ∈ parts (knows Spy evs);
         C ∉ bad;  evs ∈ set_pur|] ==> parts {X} ⊆ used evs"
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply simp_all
apply safe
apply blast+
done
```

Lemma needed below: for the case that if PRes is present, then `LID_M` has been used.

```
lemma PRes_imp_LID_used:
     "[| Crypt (priSK M) (Hash {|N, X|}) ∈ parts (knows Spy evs);
         M ∉ bad;  evs ∈ set_pur|] ==> N ∈ used evs"
by (drule signed_imp_used, auto)
```

When C receives PRes, he knows that M and P agreed to the purchase details. He also knows that P is the same PG as before

```
lemma C_verifies_PRes_lemma:
     "[| Crypt (priSK M) (Hash MsgPRes) ∈ parts (knows Spy evs);
         Notes C {|Number LID_M, Trans |} ∈ set evs;
         Trans = {| Agent M, Agent C, Number OrderDesc, Number PurchAmt |};
         MsgPRes = {|Number LID_M, Number XID, Nonce Chall_C,
                Hash (Number PurchAmt)|};
         evs ∈ set_pur;  M ∉ bad|]
  ==> ∃j KP.
        Notes M {|Number LID_M, Agent (PG j), Trans |}
```

```
          ∈ set evs &
        Gets M (EncB (priSK (PG j)) KP (pubEK M)
                {|Number LID_M, Number XID, Number PurchAmt|}
                authCode)
          ∈ set evs &
        Says M C (sign (priSK M) MsgPRes) ∈ set evs"
apply clarify
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply simp_all
apply blast
apply blast
apply (blast dest: PRes_imp_LID_used)
apply (frule M_Notes_PG, auto)
apply (blast dest: unique_LID_M)
done

lemma C_verifies_PRes:
     "[| MsgPRes = {|Number LID_M, Number XID, Nonce Chall_C,
                     Hash (Number PurchAmt)|};
         Gets C (sign (priSK M) MsgPRes) ∈ set evs;
         Notes C {|Number LID_M, Agent M, Agent C, Number OrderDesc,
                   Number PurchAmt|} ∈ set evs;
         evs ∈ set_pur;  M ∉ bad|]
  ==> ∃P KP trans.
        Notes M {|Number LID_M,Agent P, trans|} ∈ set evs &
        Gets M (EncB (priSK P) KP (pubEK M)
                {|Number LID_M, Number XID, Number PurchAmt|}
                authCode)
          ∈ set evs &
        Says M C (sign (priSK M) MsgPRes) ∈ set evs"
apply (rule C_verifies_PRes_lemma [THEN exE])
apply (auto simp add: sign_def)
done
```

6.10 Proofs for Signed Purchases

Some Useful Lemmas: the cardholder knows what he is doing

```
lemma Crypt_imp_Says_Cardholder:
     "[| Crypt K {|{|{|Number LID_M, others|}, Hash OIData|}, Hash PANData|}
           ∈ parts (knows Spy evs);
         PANData = {|Pan (pan (Cardholder k)), Nonce (PANSecret k)|};
         Key K ∉ analz (knows Spy evs);
         evs ∈ set_pur|]
  ==> ∃M shash EK HPIData.
       Says (Cardholder k) M {|{|shash,
          Crypt K
            {|{|{|Number LID_M, others|}, Hash OIData|}, Hash PANData|},
           Crypt EK {|Key K, PANData|}|},
          OIData, HPIData|} ∈ set evs"
apply (erule rev_mp)
```

```
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_pur.induct, analz_mono_contra)
apply (frule_tac [9] AuthReq_msg_in_parts_spies) — AuthReq
apply simp_all
apply auto
done
```

```
lemma Says_PReqS_imp_trans_details_C:
     "[| MsgPReqS = {|{|shash,
                 Crypt K
                  {|{|{|Number LID_M, PIrest|}, Hash OIData|}, hashpd|},
            cryptek|}, data|};
         Says (Cardholder k) M MsgPReqS ∈ set evs;
         evs ∈ set_pur |]
   ==> ∃trans.
           Notes (Cardholder k)
                 {|Number LID_M, Agent M, Agent (Cardholder k), trans|}
            ∈ set evs"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule set_pur.induct)
apply (simp_all (no_asm_simp))
apply auto
done
```

Can't happen: only Merchants create this type of Note

```
lemma Notes_Cardholder_self_False:
     "[|Notes (Cardholder k)
          {|Number n, Agent P, Agent (Cardholder k), Agent C, etc|} ∈ set
evs;
        evs ∈ set_pur|] ==> False"
apply (erule rev_mp)
apply (erule set_pur.induct, auto)
done
```

When M sees a dual signature, he knows that it originated with C. Using XID he knows it was intended for him. This guarantee isn't useful to P, who never gets OIData.

```
lemma M_verifies_Signed_PReq:
     "[| MsgDualSign = {|HPIData, Hash OIData|};
         OIData = {|Number LID_M, etc|};
         Crypt (priSK C) (Hash MsgDualSign) ∈ parts (knows Spy evs);
         Notes M {|Number LID_M, Agent P, extras|} ∈ set evs;
         M = Merchant i;  C = Cardholder k;  C ∉ bad;  evs ∈ set_pur|]
      ==> ∃PIData PICrypt.
            HPIData = Hash PIData &
            Says C M {|{|sign (priSK C) MsgDualSign, PICrypt|}, OIData, Hash
PIData|}
              ∈ set evs"
apply clarify
apply (erule rev_mp)
apply (erule rev_mp)
```

apply *(erule set_pur.induct)*
apply *(frule_tac [9] AuthReq_msg_in_parts_spies)* — AuthReq
apply *simp_all*
apply *blast*
apply *(force dest!: signed_Hash_imp_used)*
apply *(clarify)* — speeds next step
apply *(blast dest: unique_LID_M)*
apply *(blast dest!: Notes_Cardholder_self_False)*
done

When P sees a dual signature, he knows that it originated with C. and was intended for M. This guarantee isn't useful to M, who never gets PIData. I don't see how to link *PG j* and *LID_M* without assuming $M \notin$ *bad*.

```
lemma P_verifies_Signed_PReq:
     "[| MsgDualSign = {|Hash PIData, HOIData|};
         PIData = {|PIHead, PANData|};
         PIHead = {|Number LID_M, Number XID, HOD, Number PurchAmt, Agent M,
                    TransStain|};
         Crypt (priSK C) (Hash MsgDualSign) ∈ parts (knows Spy evs);
         evs ∈ set_pur;  C ∉ bad;  M ∉ bad|]
    ==> ∃OIData OrderDesc K j trans.
          HOD = Hash{|Number OrderDesc, Number PurchAmt|} &
          HOIData = Hash OIData &
          Notes M {|Number LID_M, Agent (PG j), trans|} ∈ set evs &
          Says C M {|{|sign (priSK C) MsgDualSign,
                     EXcrypt K (pubEK (PG j))
                                {|PIHead, Hash OIData|} PANData|},
                     OIData, Hash PIData|}
            ∈ set evs"
apply clarify
apply (erule rev_mp)
apply (erule set_pur.induct, simp_all)
apply (auto dest!: C_gets_correct_PG)
done

lemma C_determines_EKj_signed:
     "[| Says C M {|{|sign (priSK C) text,
                      EXcrypt K EKj {|PIHead, X|} Y|}, Z|} ∈ set evs;
         PIHead = {|Number LID_M, Number XID, W|};
         C = Cardholder k;  evs ∈ set_pur;  M ∉ bad|]
  ==> ∃ trans j.
         Notes M {|Number LID_M, Agent (PG j), trans|} ∈ set evs &
         EKj = pubEK (PG j)"
apply clarify
apply (erule rev_mp)
apply (erule set_pur.induct, simp_all, auto)
apply (blast dest: C_gets_correct_PG)
done

lemma M_Says_AuthReq:
     "[| AuthReqData = {|Number LID_M, Number XID, HOIData, HOD|};
         sign (priSK M) {|AuthReqData, Hash P_I|} ∈ parts (knows Spy evs);
         evs ∈ set_pur;  M ∉ bad|]
   ==> ∃j trans KM.
```

```
          Notes M {|Number LID_M, Agent (PG j), trans |} ∈ set evs &
            Says M (PG j)
              (EncB (priSK M) KM (pubEK (PG j)) AuthReqData P_I)
            ∈ set evs"
```
apply *(rule refl [THEN P_verifies_AuthReq, THEN exE])*
apply *(auto simp add: sign_def)*
done

A variant of *M_verifies_Signed_PReq* with explicit PI information. Even here we cannot be certain about what C sent to M, since a bad PG could have replaced the two key fields. (NOT USED)

lemma *Signed_PReq_imp_Says_Cardholder:*
```
     "[| MsgDualSign = {|Hash PIData, Hash OIData|};
         OIData = {|Number LID_M, Number XID, Nonce Chall_C, HOD, etc|};
         PIHead = {|Number LID_M, Number XID, HOD, Number PurchAmt, Agent M,
                    TransStain|};
         PIData = {|PIHead, PANData|};
         Crypt (priSK C) (Hash MsgDualSign) ∈ parts (knows Spy evs);
         M = Merchant i;  C = Cardholder k;  C ∉ bad;  evs ∈ set_pur|]
      ==> ∃KC EKj.
           Says C M {|{|sign (priSK C) MsgDualSign,
                      EXcrypt KC EKj {|PIHead, Hash OIData|} PANData|},
                      OIData, Hash PIData|}
             ∈ set evs"
```
apply *clarify*
apply *(erule rev_mp)*
apply *(erule rev_mp)*
apply *(erule set_pur.induct, simp_all, auto)*
done

When P receives an AuthReq and a dual signature, he knows that C and M agree on the essential details. PurchAmt however is never sent by M to P; instead C and M both send *HOD = Hash {|Number OrderDesc, Number PurchAmt|}* and P compares the two copies of HOD.

Agreement can't be proved for some things, including the symmetric keys used in the digital envelopes. On the other hand, M knows the true identity of PG (namely j'), and sends AReq there; he can't, however, check that the EXcrypt involves the correct PG's key.

lemma *P_sees_CM_agreement:*
```
     "[| AuthReqData = {|Number LID_M, Number XID, HOIData, HOD|};
         KC ∈ symKeys;
         Gets (PG j) (EncB (priSK M) KM (pubEK (PG j)) AuthReqData P_I)
           ∈ set evs;
         C = Cardholder k;
         PI_sign = sign (priSK C) {|Hash PIData, HOIData|};
         P_I = {|PI_sign,
                 EXcrypt KC (pubEK (PG j)) {|PIHead, HOIData|} PANData|};
         PANData = {|Pan (pan C), Nonce (PANSecret k)|};
         PIData = {|PIHead, PANData|};
         PIHead = {|Number LID_M, Number XID, HOD, Number PurchAmt, Agent M,
                    TransStain|};
         evs ∈ set_pur;  C ∉ bad;  M ∉ bad|]
  ==> ∃OIData OrderDesc KM' trans j' KC' KC'' P_I' P_I''.
```

```
          HOD = Hash{|Number OrderDesc, Number PurchAmt|} &
          HOIData = Hash OIData &
          Notes M {|Number LID_M, Agent (PG j'), trans|} ∈ set evs &
          Says C M {|P_I', OIData, Hash PIData|} ∈ set evs &
          Says M (PG j') (EncB (priSK M) KM' (pubEK (PG j'))
                         AuthReqData P_I'') ∈ set evs &
          P_I' = {|PI_sign,
            EXcrypt KC' (pubEK (PG j')) {|PIHead, Hash OIData|} PANData|}
&
          P_I'' = {|PI_sign,
            EXcrypt KC'' (pubEK (PG j)) {|PIHead, Hash OIData|} PANData|}"
apply clarify
apply (rule exE)
apply (rule P_verifies_Signed_PReq [OF refl refl refl])
apply (simp (no_asm_use) add: sign_def EncB_def, blast)
apply (assumption+, clarify, simp)
apply (drule Gets_imp_knows_Spy [THEN parts.Inj], assumption)
apply (blast elim: EncB_partsE dest: refl [THEN M_Says_AuthReq] unique_LID_M2)
done

end
```